SPARQL distributed query processing over linked data

Abdoul Macina

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

SPARQL Distributed Query Processing over Linked Data

Traitement de requêtes SPARQL sur des données liées

Abdoul MACINA

laboratoire I3S, CNRS UMR-7271, équipe SPARKS

Présentée en vue de l'obtention du grade de docteur en informatique **d'**Université Côte d'Azur

Dirigée par : Johan Montagnat, Directeur de Recherche, CNRS

Co-dirigée par : Olivier Corby, Chargé de Recherche, INRIA

Soutenue le : 17/12/2018

Devant le jury, composé de :

Président: Andrea Tettamanzi, Professeur, Université de Nice Sophia Antipolis

Rapporteurs: Esther Pacitti, Professeur, Université de Montpellier

Hala Skaf-Molli, Maître de Conférences, Université de Nantes

Examinateurs: Oscar Corcho, Professor, Universidad Politécnica de Madrid

UNIVERSITÉ CÔTE D'AZUR

LABEX UCN User Centric Network @ SOPHIA

INVESTISSEMENTS D'AVENIR

Abstract

Driven by the Semantic Web standards, an increasing number of RDF data sources are published and connected over the Web by data providers, leading to a large distributed linked data network. However, exploiting the wealth of these data sources is very challenging for data consumers considering the data distribution, their volume growth and data sources autonomy. In the Linked Data context, federation engines allow querying these distributed data sources by relying on Distributed Query Processing (DQP) techniques. Nevertheless, a naive implementation of the DQP approach may generate a tremendous number of remote requests towards data sources and numerous intermediate results, thus leading to costly network communications. Furthermore, the distributed query semantics is often overlooked. Query expressiveness, data partitioning, and data replication are other challenges to be taken into account.

To address these challenges, we first proposed a Distributed Query Processing semantics which preserves the SPARQL language expressiveness. Afterwards, we presented several strategies for a federated query engine that transparently addresses distributed data sources. Firstly, we proposed a federated query semantics on top of RDF and SPARQL standards, and following this, we specified the semantics of federated SPARQL queries on top of the standard SPARQL semantics through a set of rewrite rules relying on service clause. Secondly, we proposed both static and dynamic strategies for the main steps of DQP approach in order to allow transparent and efficient distributed and autonomous data sources querying, and therefore enhance the federated query processing. The static optimizations rely on the results of the source selection step in which we proposed a Sampling Query-Based approach. On the one hand, based on these results we proposed a Hybrid BGP-Triple query rewriting approach which addresses both horizontal and vertical data partitions and reduces the query engines workload. On the other hand, we introduced a static query sorting approach which combines the cost estimation, heuristics on query patterns and query expressions links. The dynamic optimizations are achieved during query evaluation step at three levels. First, the bindings (already known values of variables) are propagated to the following sub-queries sharing the same variables. Secondly, we use parallelism to concurrently query remote data sources and independent remote requests. Finally, we use triple results duplicate-aware

evaluation to avoid results redundancy which are side effects of triple replication. We implemented and evaluated our approach and optimization strategies in a federated query engine to prove their applicability.

Keywords: Semantic Web, Web of Data, Linked Data, Linked Open Data, Data Integration, Distributed Query Processing, Federated query evaluation, SPARQL, Query Optimization

Résumé

De plus en plus de sources de données liées sont publiées par des fournisseurs de données à travers le Web en s'appuyant sur les technologies du Web sémantique, formant ainsi un large réseau de données distribuées. Cependant, il devient plus difficile pour les consommateurs de données de tirer profit de la richesse de ces données, compte tenu de leur distribution, de l'augmentation de leur volume et de l'autonomie des sources de données. Dans le contexte du Web des données, les moteurs fédérateurs de données permettent d'interroger des sources de données distribuées en utilisant des techniques de traitement de requêtes distribuées. Cependant, une mise en œuvre naïve de ces techniques peut générer un nombre considérable de requêtes distantes vers les sources de données et de nombreux résultats intermédiaires entraînant ainsi un long temps de traitement des requêtes et une communication réseau coûteuse. Par ailleurs, la sémantique des requêtes distribuées n'est pas clairement définie. L'expressivité des requêtes, le partitionnement des données et la réplication des données sont d'autres défis auxquels doivent faire face les moteurs de requêtes.

Pour répondre à ces défis, nous avons d'abord proposé une sémantique de traitement de requêtes distribuées qui préserve l'expressivité du langage SPARQL et nous avons présenté plusieurs stratégies d'optimisation pour un moteur de requêtes fédérées qui interroge de manière transparente les sources de données distribuées. Notre sémantique de requêtes distribuées s'appuie sur les standards RDF et SPARQL. Elle est mise en œuvre à travers un ensemble de règles de réécriture basées sur la clause SERVICE. Des stratégies d'optimisation statiques et dynamiques proposées sont pour chacune des principales étapes du processus de traitement de requêtes distribuées afin de permettre une interrogation efficace et transparente des sources de données à la fois distribuées et autonomes et par conséquent améliorer la gestion des requêtes fédérées. Les optimisations statiques portent sur la sélection des sources et le tri des requêtes. Une approche hybride de réécriture de requêtes traite à la fois les partitions de données horizontales et verticales et réduit la charge de travail des moteurs de requêtes. Une approche statique de tri des requêtes combine l'estimation des coûts, des heuristiques sur les modèles de requêtes et les liens entre les requêtes. Les optimisations dynamiques sont effectuées lors de l'étape d'évaluation des requêtes à trois niveaux. D'abord, les variables dont les valeurs sont déjà connues sont propagées aux sous-requêtes suivantes partageant les mêmes variables. Ensuite, nous avons utilisé le parallélisme pour interroger simultanément des sources de données distantes et envoyer des requêtes indépendantes. Enfin, nous avons mis en œuvre une évaluation de requêtes qui prend en compte la redondance des données pour éviter dans les résultats des doublons induits uniquement par cette redondance liée à la distribution. Nous avons implémenté et évalué les différentes

stratégies d'optimisation dans un moteur de requêtes fédérées pour montrer son applicabilité.

Mots-clés:Web de Données, Données liées, Intégration des données, Traitement de requêtes distribuées, Evaluation des requêtes fédérées, Optimisation de requêtes SPARQL.

Contents

1 Introduction **1**
- 1.1 Context . . . 2
- 1.2 Motivations . . . 3
- 1.3 Objectives . . . 4
- 1.4 Thesis outline . . . 4

2 From Data Integration to Linked Open Data **7**
- 2.1 Introduction . . . 8
- 2.2 Data integration principles . . . 8
  - 2.2.1 Materialized data integration . . . 9
  - 2.2.2 Virtualized data integration . . . 10
  - 2.2.3 Virtual data integration mapping . . . 11
- 2.3 Data distribution and data fragmentation . . . 11
- 2.4 Linked Open Data integration . . . 13
  - 2.4.1 Linked Open Data principles . . . 13
  - 2.4.2 Resource Description Framework . . . 14
  - 2.4.3 The SPARQL query language . . . 16
  - 2.4.4 Linked Open Data (LOD) integration . . . 19

3 Federated query processing: State-of-the-Art **21**
- 3.1 Introduction . . . 22
- 3.2 Distributed Query Processing principle . . . 22
- 3.3 Federated SPARQL query processing engines . . . 23
  - 3.3.1 Source selection . . . 25
  - 3.3.2 Query rewriting . . . 27
  - 3.3.3 Query planning . . . 29
  - 3.3.4 Query evaluation . . . 31
- 3.4 Discussion and Challenges . . . 33
- 3.5 Conclusion . . . 35

4 Towards Federated SPARQL query semantics and specifications **37**
- 4.1 Introduction . . . 38
- 4.2 SPARQL query evaluation over a distributed knowledge graph . . . 38
- 4.3 Federated dataset semantics on top of RDF semantics . . . 42

4.4 SPARQL features over distributed data sources . . . 43
4.4.1 Rewrite rules . . . 44
4.4.2 Triples replication . . . 48
4.5 Conclusion . . . 49

5 Towards efficient Federated SPARQL Query Processing 51
5.1 Introduction . . . 52
5.2 Hybrid sources selection . . . 52
5.2.1 Sample source selection . . . 53
5.2.2 Source selection algorithm . . . 55
5.3 Hybrid query rewriting . . . 57
5.3.1 Global join decomposition . . . 61
5.3.2 Joins generation strategy . . . 61
5.3.3 Query rewriting algorithm . . . 63
5.3.4 Hybrid rewriting expressions evaluation . . . 69
5.4 Sub-queries sorting optimization . . . 72
5.4.1 Cost estimation algorithm . . . 72
5.4.2 Cost-based and shared links sorting approach . . . 74
5.5 Query evaluation . . . 76
5.5.1 Bindings strategy . . . 76
5.5.2 Parallelism strategy . . . 76
5.6 Conclusion . . . 77

6 Implementation and Evaluation 79
6.1 Introduction . . . 80
6.2 Implementation . . . 80
6.3 Evaluation . . . 82
6.3.1 Hybrid Data Partitioning (HDP) experiment . . . 82
6.3.2 FedBench benchmarck . . . 89
6.3.3 KGRAM-DQP vs FedX . . . 96
6.4 Conclusions . . . 100

7 Conclusions and Perspectives 101
7.1 Summary . . . 102
7.2 Perspectives . . . 104

Bibliography 107

List of Figures

2.1 Data integration approaches 10
2.2 Statement in RDF 15
2.3 Example of RDF graph 16
2.4 Linking Open Data cloud diagram 20

3.1 SPARQL DQP steps 23
3.2 SPARQL DQP engines architecture 24

4.1 Data replication on Virtual Knowledge Graph 40
4.2 Side effect of data replication 41

6.1 KGRAM-DQP architecture 81
6.2 Top: query processing time. Bottom: remote requests sent 88
6.3 Optimized KGRAM-DQP - FedX: query processing time 94
6.4 Optimized KGRAM-DQP - FedX (first) - FedX (+caching): query processing time 95
6.5 Optimized KGRAM-DQP and FedX performance 98
6.6 Number of results retrieved on INSEEE data 99

List of Tables

3.1 Source selections approaches for SPARQL DQP engines 27
3.2 Query Rewriting approaches vs data partitioning 29
3.3 Query Planning approaches for SPARQL DQP engines 31
3.4 Query Evaluation approaches for SPARQL DQP engines 33

5.1 Sample data sources for hybrid rewriting 60
5.2 Evaluation operators mappings results 60

6.1 Data sets partitioning 83
6.2 FedBench Datasets 89
6.3 Number of results on FedBench 95
6.4 Number of results 99

1 Introduction

Contents

1.1 Context 2
1.2 Motivations 3
1.3 Objectives 4
1.4 Thesis outline 4

1.1 Context

The Semantic Web is an extension of the Web, invented by Tim Berners-Lee in 1989 [9], to move from a Web of documents to a Web of data. The traditional Web is more dedicated to humans as it allows persons to share and to have access to information by navigating in a network of documents connected through hypertext links. However, the information of this network of documents without explicit semantics is meaningless for machines. Furthermore, the evolution of the Web over the three last decades resulted in a an enriched network which links large amount of data, things, applications, etc, in addition to documents. As a consequence, achieving processing on this network became very challenging hence the need to make the Web more computer-processable. Tim Berners-Lee et al. proposed in 2001 [11] the Semantic Web as an extension of the current Web in which the information has a well-defined meaning (structured data and semantics) and therefore is readable for both machines and humans and helping them to work in cooperation.

To enable an efficient processing of the Semantic Web, the data should be shared and connected through links so that for a given data, related data can be easily found, hence the term "Linked Data". In 2006, the Linked Data principles [10] are defined by Tim Berners-Lee. He also proposed a set of rules which enable data providers to publish and to link data to other related data. As a result, the number of linked data sources over the Web increased [13] by relying on URI [8] and on a set of W3C standards [77] (RDF [68], RDFS [69], SPARQL [78] and OWL [57]).

- URI (Uniform Resource Identifier): is used to identify things.
- RDF (Resource Description Framework): is used as data model for semantic data representation.
- RDFS (RDF Schema): is a basic schema language for RDF data.
- SPARQL (SPARQL Protocol and RDF Query Language): is both a query language and a protocol to query RDF data sources over the Web. The language allows users to query RDF data and provides the protocol to send HTTP requests to remote data source servers.
- OWL (Web Ontology Language): is a richer and more complex schema language used to define ontologies and to perform sophisticated reasoning on data.

Thus, data providers make their data available through RDF datasets and data consumers mainly get access to them through SPARQL endpoints. The growth of the volume of data and the number of data sources gradually led the Web of data to a large network of distributed and connected knowledge bases at the scale of the Web. It becomes thus challenging to query these datasets while addressing both data distribution and data volume. Hence, distributed and federated querying becomes a prominent research question in the data processing area [46, 31].

1.2 Motivations

Federated querying consists in processing a query over a set of distributed data sources through their SPARQL endpoints by rewriting an initial query in a set of sub-queries and sending them to the appropriate source. The SPARQL standard addresses federated query processing through the SERVICE clause which allows to send a query to a given data source by specifying its URI. However, it requires for the query designer to know beforehand how knowledge pieces matching the initial query are distributed over the data sources to write proper SERVICE clauses.

Federated query processing allows users to transparently query distributed data sources by automating this process through a federated query engine. To do so, the federated engine first needs to query the data sources and identify the ones capable to answer the parts of the query, called relevant sources. This process is performed in a source selection step by sending SPARQL queries to endpoints. Afterwards, the initial user query is rewritten to suit the identified relevant sources and to allow an optimal evaluation.

Linked Data Fragment (LDF) [39, 88] is an initiative as an alternative approach to SPARQL endpoints. LDF aims at addressing the availability issue of data sources which prevents reliably querying these data. Thus, to balance the workload between the data sources servers and the clients and to increase the data availability, a fragment of needed data is loaded at the client side based on a specific selector to allow a local processing. LDF, and more specifically Triple Pattern Fragment(TPF) [87] based on it is out of the scope of this thesis.

With respect to federated query processing in the Web of data context, as shown in Chapter 3, several approaches have been proposed to address query processing optimization issues. However, performance and scalability issues rapidly arise due to the data distribution and volume. Moreover, the federated engines results can vary depending on their specific interpretation of the distributed query semantics. Finally,

in most of the work in the state-of-the-art, the query expressiveness is reduced to a subset of SPARQL features.

1.3 Objectives

In this thesis, we tackle the federated query processing in a context of autonomous data sources. We first aim at improving the federated query expressiveness by making it more compliant with SPARQL. The second goal is to enhance the reliability on query results by defining a clear federated query semantics. Finally, the third objective is to propose an efficient federated query approach that allows to transparently query distributed data sources while taking into account the data partitioning, data replication and results completeness challenges. This work relies on a previous work that proposed the KGRAM-DQP SPARQL federated query engine [30].

More specifically, this thesis will address the following main research question:

- RQ_1: How to specify a semantics for SPARQL federated queries?
- RQ_2: How to allow highly expressive federated queries in the context of Linked Data?
- RQ_3: How to improve the federated query engines performance ?

  We break down this research question into several sub-questions:

  - RQ_{3_a}: How to reduce processing time and communication cost ?
  - RQ_{3_b}: How to ensure the results completeness?
  - RQ_{3_c}: How to address the data partitioning in this process?

1.4 Thesis outline

- In Chapter 2, we introduce the general principles of data integration through the mappings approaches used to process it. Afterwards, we review the data fragmentation approaches and we show their impact on data integration. Then we focus on data integration in the Linked Open Data context after describing its principles and the data representation and querying.

- Chapter 3 reports the state of the art for federated query processing approach over Linked Open Data. We first describe the Distributed Query Processing principle through its main steps: source selection, query rewriting, query planning and query evaluation. Afterwards, we introduce several SPARQL federated query processing engines, while describing their optimization approaches for each step. Finally, we discuss these optimizations and underline challenges to improve SPARQL federated query processing performance.

- In Chapter 4, we investigate the SPARQL federated query semantics by comparing the evaluation of a standard SPARQL query over a centralized and a distributed RDF graph to highlight the necessity to define a clear semantics for SPARQL federated queries. Based on this comparison, we express the constraints that federated query engines should address to be compliant with both RDF and SPARQL standards. Finally, we express SPARQL federated queries through the standard SPARQL features, using the SERVICE clause, in order to define their semantics on top of standard SPARQL semantics.

- In Chapter 5, we propose optimization strategies for each step of the distributed query processing. For source selection step we introduce a sampling query-based approach which aims at identifying relevant sources and retrieving triple patterns cardinality. Afterwards, we propose a query decomposition technique that aims at pushing as much of the initial query as possible towards the remote data sources to reduce the processing cost at query engine side. Then, we present a query sorting approach based on cardinality cost estimation and shared variables heuristics. For the query evaluation, we introduce a results redundancy aware query evaluation approach to address side effect of triples replication on distributed data sources.

- Chapter 6 reports the experimental results to assess the performance of our federated query processing approach and optimizations. To do so, we first present KGRAM-DQP federated query engine that supports the implementation. Then, we present the results for the different experiments performed and we discuss the impact of our strategy regarding several challenges, namely results completeness, data replication and query processing efficiency.

2 From Data Integration to Linked Open Data

Contents

2.1 Introduction **8**
2.2 Data integration principles **8**
2.2.1 Materialized data integration 9
2.2.2 Virtualized data integration 10
2.2.3 Virtual data integration mapping 11
2.3 Data distribution and data fragmentation **11**
2.4 Linked Open Data integration **13**
2.4.1 Linked Open Data principles 13
2.4.2 Resource Description Framework 14
2.4.3 The SPARQL query language 16
2.4.4 Linked Open Data (LOD) integration 19

2.1 Introduction

In several areas of activity such as science, economics or research, sharing and integrating information and knowledge are needed. However, information and knowledge are often distributed in several sources. Data integration aims at giving access through a unified view to data located in miscellaneous sources. For instance, search engines enable users to query several data sources through a single interface instead of querying each data source.

Previously, this integration was mainly done on data internal to institutions and structures, e.g. when merging two company department databases. Thus, different sources are integrated in a common source to materialize the integrated database. Later, in particular with the emergence of the Web, links between independent data sources became common. In this context, data sources autonomy prevents their clustering. To face this challenge, another data integration approach, called virtual integration has been proposed. It consists in federating autonomous sources through a virtual unified view.

More generally, virtual data integration concerns all distributed data sources. Indeed, for the sake of efficiency, reliability and availability, data can be intentionally distributed: fragmented (split up in several partitions) and allocated (assigned to different databases). On the other hand, the data can also be replicated (duplicating data in another database) and databases interconnected through the use of shared schemas or ontology to describe and produce data on related objects and concepts. Across the Web, interconnected and distributed data sources are called Linked Open Data.

In this chapter, we first review the general principles of data integration, the mapping approaches used to manage different conceptual schemas and the methods by which the integration is implemented in practice. Then, we underline the different data fragmentation approaches and we show their impact on the data integration process. Afterwards, we focus on the aspects of Linked Open Data (LOD) which are relevant for our work. We outline Linked Open Data principles. Then, we review semantic data representation and querying. Finally, we describe the LOD integration process.

2.2 Data integration principles

Data integration [34, 36] enables to transparently manage several distributed data sources (or databases) by providing a unified view over all data. In, other words,

the user should be able to query the data sources without dealing with the data distribution and possible replication. We define *distributed data sources* as a set of autonomous databases distributed over a network but logically related.

Formally, a data integration [15] system is defined as a triple {*G, S, M* } where:

- G is the global conceptual schema of the unified view,
- S is the set of sources structured along local schemas $\{S_1, ..., S_n\}$
- M is the mapping between G and S which links global and local schemas.

The effective integration of distributed data sources can be either physical or logical [58]. In the former approach, the unified view is materialized in a distinct database in which the different data sources are aggregated. In the latter, the unified view is virtual and data are kept in their original databases. Instead, the data integration is virtually handled through a system managing the global schema and the data mappings to express queries sent to the local schemas.

2.2.1 Materialized data integration

The *materialized* data integration approach, also called *data warehousing* [33, 35] or *Extract-Transform-Load (ETL)* approach, consists in centralizing data in one database. Therefore data is queried in a centralized way. As suggested by the term ETL we can identify three phases. The extraction phase uploads the data from the target sources. Then, during the transformation phase, extracted datasets are transformed to obtain homogeneous data (aligned to the global schema G). Afterwards, these datasets are aggregated and are saved in the data warehouse through the loading step.

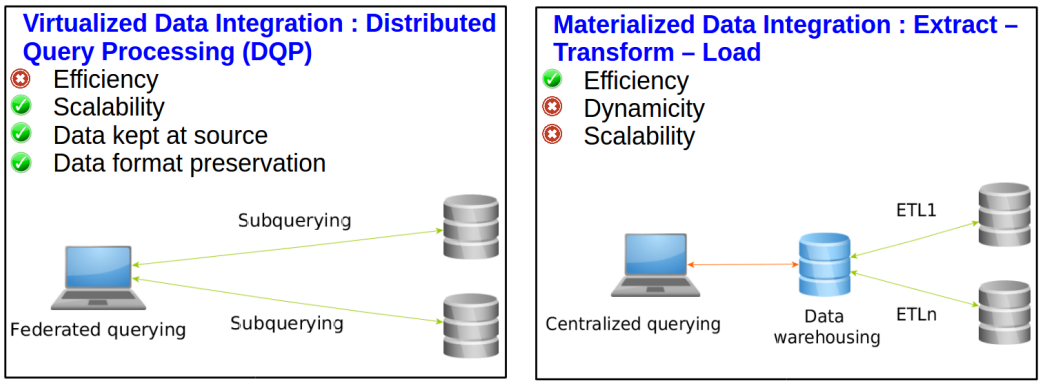
The main drawbacks of the materialization approach are (i) the need to update the central repository (repeat the ETL process) when remote sources evolve or a new data source appears, and (ii) to duplicate every data from the source repositories. This approach has limited scalability and data freshness issues. It is also not suitable in the context of autonomous and distributed data involving managing sensible data, hardly relocatable for ethical or legal reasons. However, it has the advantage of reducing data communication after the ETL process and making available all data from one dataset. It may be very efficient for query processing. In particular, if there are few changes in data sources or using slightly outdated data is allowed.

2.2.2 Virtualized data integration

The *virtualized* data integration is also known as *federated querying* or *distributed query processing (DQP)*. Based on a query answering [37] interface, handled by a federation engine, the virtual data integration system provides answers to queries expressed in the global schema by translating them into the local schemas through the mapping M.

The data sources distribution is addressed through query rewriting mechanisms. The initial query is split in several sub-queries which are sent to remote data sources for evaluation. Then, the intermediate results generated by these sub-queries are gathered through joins or unions by the federated query engine. This approach prevents the replication and the periodical extraction of the source databases. Furthermore, new data sources can be added more easily in the integration process. The datasets are kept at initial sources and the data formats are preserved. However the DQP approach is also difficult to achieve efficiently. Indeed, this approach requires handling the physical distribution, the network communication cost, data heterogeneity, and unavailability or failure of data sources. The Figure 2.1 summarizes the advantages and the drawbacks for each approach.

Figure. 2.1: Data integration approaches



In addition, the mapping M deals with the possible heterogeneity of distributed data sources, without doing an actual transformation of data, by using the global schema as a homogeneous model over multiple data sources. Consequently the mapping is no longer necessary. Two main approaches have been proposed to tackle the virtual data integration mapping expression: *Global-As-View (GAV)* and *Local-As-View (LAV)* [48, 49].

2.2.3 Virtual data integration mapping

In the GAV approach, the global schema is defined as a view over the data sources (union of views over each local schema). It is a bottom-up approach, given that the global schema is generated from the local schemas of data sources. The main advantage of GAV is a straightforward query processing. Indeed, to answer a query, a concept of the global schema has to be replaced by the corresponding one(s) in the local schemas. Moreover, the global schema update can be easily done since views over local sources are independent. The GAV approach also has some downsides: (i) the need to know all local schemas to be able to build the global mapping, and (ii) adding a new local schema may imply modifying several global concepts.

Conversely, the LAV is a top-down approach. It represents each local schema as a view of the global schema. In fact, the global schema is defined and each local schema is expressed based on it. The main advantage of the LAV approach is to allow gracefully increasing the number of sources while preserving the global schema. Besides, the independence of data sources in LAV systems facilitates handling data sources unavailability. However, the LAV global schema is less scalable since changing a concept in the global schema may involve modifying the mapping of several local schemas. In addition, the LAV query rewriting process is very costly [52] arising from the complexity of rewriting a query expressed in the global schema into all local schemas.

Furthermore, a third approach, called *Global-and-Local-As-View (GLAV)*, has been proposed to take advantage of both approaches [29]. The GLAV approach expresses both a global schema and local schemas by using LAV and GAV query rewriting processes to reduce the LAV complexity while ensuring the global schema scalability.

Thus, with regards to virtual integration mapping expression, the GAV approach is more scalable and provides a simple query answering process, while the LAV approach is more appropriate in a context of dynamic local data sources with the drawback of query processing complexity. The GLAV approach is a middle way between the two foregoing strategies. Therefore the best mapping expression depends on data sources schemas which are related to data distribution and to data fragmentation.

2.3 Data distribution and data fragmentation

Data fragmentation, alternatively called data partitioning, consists in separating data in several partitions and storing them in different sites for the sake of query

processing efficiency and reliability. In the literature, we distinguish two main data fragmentation approaches: horizontal partitioning [24] and vertical partitioning [18, 1]. We exemplify below data fragmentation approaches with relational databases on which they have first been implemented. Horizontal data partitioning consists in splitting relations along their tuples. Thus, to each partition is assigned a subset of tuples (rows). Vertical data partitioning, on the other hand, consists in fragmenting the initial relations throughout their attributes. Each partition contains a subset of attributes (columns) of the relations. In addition, there is a mixed approach [3] which combines horizontal and vertical partitioning. When data fragmentation is performed, three criteria have to be complied with to ensure correctness:

- *completeness* [59, 38]: The fragmentation completeness ensures no data loss during the decomposition process between the initial relation and the subset of relations in partitions

- *disjointness*: The disjointness avoids data overlapping between the fragments.

- *reconstruction*: The initially fragmented relation should be reformable through the different fragments. The data for horizontal and vertical partitions are reconstructed across union or join operators respectively. For instance, for a relation R fragmented into $F_R = \{R_1, R_2, ..., R_n\}$, the reconstruction for horizontal and vertical partitions are formally defined as follows respectively:

  - $R = \cup R_i$

  - $R = \bowtie R_i$

Fragmented data is queried by translating the query expressed on the global relation into several sub-queries expressed on the local relations of the partitions. To some extent, this is similar to the LAV approach. Indeed, query answering over distributed data sources amounts to integrate fragmented data in addition to the complexity related to the data distribution, data replication and communication cost.

When centralized data is intentionally fragmented, the more suitable partition approach may be selected based on several criteria like most queried data or most common joins. However, in the LOD data aggregation context, due to the data sources autonomy the data may be both vertically and horizontally partitioned. In addition, it is also common to face partially or fully replicated data in the distributed context, either to improve their availability, or because the fragmentation process is not managed. All these factors make the distributed databases query processing complex.

2.4 Linked Open Data integration

2.4.1 Linked Open Data principles

The *Linked Open Data* [10] results from *Open Data* and *Linked Data* concepts. Open Data[1] is data (common knowledge or information) that is available for anyone to use, modify and share. The Open data movement, also called Open Knowledge, advocates free data access under open licence. However, the data provenance should be indicated and data openness must be preserved. The whole aim behind this concept is to allow breakthrough innovation through data sharing by finding various areas of application or by cross-analyzing data to produce new or more complete data. Consequently, open data, whether it is raw or well structured, should be public. Thus, the Web has become the suitable and simple way to share open data in several domains like scientific research, life science and government data. The *Linked Data* [10] or Web of data, on the other hand, aims at connecting related data available over the Web. The goal is to build links between these data to ease data exploration by both humans and machines. Indeed, building links between different data creates a global network of data which enables users to have access to all connected and related data from a given piece of data, thus transforming the Web into a global data space [41]. The Linked Data, introduced by Tim Berners-Lee in 2006 [10], advocates several Web standards: URI (Uniform Resource Identifier) [86, 8], HTTP (Hypertext Transfer Protocol), RDF (Resource Description Framework) and SPARQL (SPARQL Protocol and RDF Query Language) to identify, describe, connect, and query data on the Web. A data provider should therefore use:

- URIs to identify any thing (all objects and concepts)
- HTTP protocol to enable things to be looked up
- RDF to describe useful information on URIs and SPARQL to query them
- RDF links between URIs in data description to provide connected data sources and to allow to discover more things

Relying on the previous principles, more and more linked data sources have been made available over the last decade, hence the use of the term "Web of Data" by analogy with the Web of documents [12]. The combination of Open and Linked data gave rise to Linked Open Data (LOD). LOD is defined as "Linked data released under an open licence, which does not impede its reuse for free" by Tim Berners-Lee. He

[1] http://opendefinition.org/od/2.1/en/

also proposed a 5-stars rating scale[2] to help data publishers to facilitate the use of their data by people, and therefore making them more powerful:

1. Data is available on the Web (whatever the format), with an open licence to be Open Data.

2. Above + in a machine-readable structured format.

3. Above + in a non-proprietary format.

4. Above + open standards from W3C (URI and RDF) are used to name, represent and query data.

5. Above + data are linked to other people's data to provide context.

2.4.2 Resource Description Framework

The Resource Description Framework (RDF) [68] is the W3C (World Wide Web Consortium) recommendation language commonly used for representing knowledge and information on the Semantic Web. The term "Resource" refers to everything that can be identified on the Web and represented through URIs. "Description" refers to the description of characteristic of resources and relations between those resources. "Framework" refers to the RDF data model as a whole with both the language and the different concrete syntaxes to represent RDF data (RDF/XML, Turtle, etc.). URIs are used either to denote concrete, or abstract concepts, e.g. events, animals, places, universe of discourse, or relations between those concepts like "Taj Mahal is located in India" (mausoleum-located-country). URIs thus aim at uniquely identifying resources on the Web in order to avoid ambiguity among them. URIs are therefore the basis for building the semantic Web by enabling to identify resources and linking them.

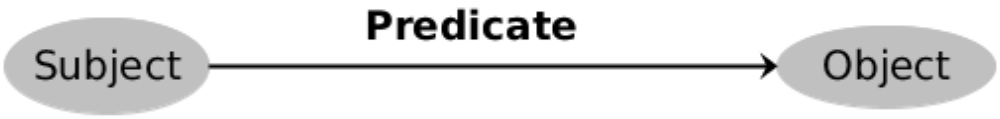
IRIs are a generalization of URIs which are a generalization of URLs (Uniform Resource Locators). URLs, commonly called Web addresses, enable to identify, to locate and to retrieve resources existing on the Web, e.g. Web page, picture or files. URL is a particular type of URI and may be thus used as URI in RDF data. URIs also enable to identify on the Web resources existing outside of the Web. For instance, we can give an URI to the Eiffel Tower and describe it through several properties (location, tip, date of construction, etc.). Finally, IRIs allow to define URIs in all languages of the world, no longer only in ASCII characters in which URIs are limited, e.g. by using Arabic or Chinese characters.

[2] http://5stardata.info/en/

Relying on IRIs, RDF describes information about the resources by expressing simple statements in the form (Subject, Predicate, Object) as depicted in Figure 2.2:

- a subject representing the resource to describe. Subjects are IRIs (Internationalized Resource Identifiers) or blank nodes.
- a predicate representing a type of property applicable to this resource. Predicates are IRIs.
- an object, representing the value of the property for a subject (this object may be a subject or object for statements). Objects are IRIs, blank nodes or literals.

Figure. 2.2: Statement in RDF

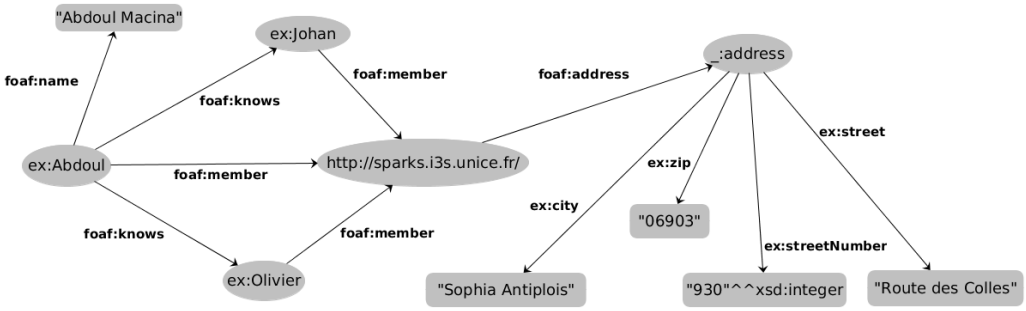


Thus, RDF is a graph model (more specifically, a directed labeled graph) composed by a set of triples (subject, predicate, object) where subjects and objects are nodes and predicates are edges linking subjects and objects. As illustrated in Figure 2.3, both nodes and edges are labeled with identifiers (IRIs or literals) in order to differentiate them. IRIs, blank nodes, and literals are collectively called RDF Terms.

Blank nodes, also called bnodes, are used to represent anonymous or unknown resources. Blank nodes are neither URIs nor literals but local anonymous resources. Their scope is limited to the RDF store in which they are defined. Consequently, blank nodes do not foster the global graph building by preventing to reuse and to refer to the same resources in different RDF sources. Therefore, in practice, they are used to describe multi-component structures like RDF lists, RDF containers and to represent complex attributes for which properties are known but not the identifier. Blank nodes can also be used to describe RDF data provenance and to protect sensitive data.

In addition to URIs and blank nodes, objects can also be literals. Literals are representing values such as strings, numbers, dates, times, etc. They can be typed (*typed literals*) or untyped (*plain literals*). The type of values is usually necessary to allow applications to know how to interpret data. For instance, if they are typed as number "6" and "06" refers to the same numerical value but two text strings are different. Plain literals are interpreted as strings.

Figure. 2.3: Example of RDF graph



RDF triples may belong to named graphs, which correspond to different (possibly overlapping) RDF data sub-graphs. RDF triples without explicit named graph attachment belong to the default graph. As RDF resources are identified by IRIs and these IRIs may be shared by different data sources referencing the same data element, RDF graphs can easily be distributed over different sources. Links between several sources are implicitly expressed by shared IRIs. The RDF global graph, connecting all graphs through shared IRIs, is sometimes referred as the global giant graph (GGG) or Linked data Cloud.

2.4.3 The SPARQL query language

The SPARQL Protocol and RDF Query Language (SPARQL) is the W3C standard language and protocol to query RDF data [78]. Triple patterns are triples where subject, predicate and/or object value can be unknown and replaced by a variable (traditionally identified by a question mark, e.g. ?v for variable v). A triple pattern may match different triples in a given RDF graph, hence its name. For instance, the triple pattern ?s p ?o matches all triples with the predicate p, and triple pattern "?s ?p ?o" matches any triple. SPARQL uses triple patterns to query RDF data. The triple patterns evaluation returns mappings between the query variables and their matching values.

A SPARQL query is a graph pattern (set of triple patterns) which has to match a sub-graph of the queried RDF graph to provide results. The graph pattern matching is performed through the conjunctions and the disjunctions of triple patterns which respectively refer to the logical "and" and "or" operators. A set of triple patterns is called a *Basic Graph Patterns (BGP)*. A BGP may either be required or optional.

SPARQL queries are composed of two main *clauses*: the first one specifies the query form, while the second is the *WHERE* clause.

There are four forms of SPARQL queries:

- The SELECT form returns a list of variable bindings where variables are bound to values (resources, literals or blank nodes) matched by the query pattern.

- The CONSTRUCT form builds an RDF graph by replacing the query pattern variables with their corresponding values for each query solution.

- The ASK form returns a boolean to indicate whether the query produces a result or not.

- The DESCRIBE form enables to retrieve the description of query pattern resources found in a form of RDF graph.

The *WHERE* clause specifies a graph pattern to be matched by queried graph resources. The *WHERE* block is mandatory for all queries forms, except for *DESCRIBE* queries, which can be directly used with the URIs of resources (e.g. DESCRIBE <http://example.com/>).

Let us consider a data set representing an organization composed by some *teams* divided into *groups*. Each group has a number of *members*. Consider the example SPARQL query Q 2.1 below:

Query Example 2.1: SPARQL Query example

```
1 prefix ns: <http://examples.fr/team#>
2 select ?team ?group ?name ?members
3 where {
4  ?team  ns:team "SPARKS".
5  ?team  ns:group ?group.
6  ?group ns:name ?name.
7  ?group ns:members ?members.
8  }
```

This query is composed of a BGP with 4 *triple patterns* (TPs) in lines 4 to 7, defining different query patterns for the RDF graph triples, and implicit join operations. Applying Q 2.1 to a single data source containing all data will return the *group*, *name* and *number* of *members* for each group in the *"SPARKS"* team in the database.

Besides BGPs, more complex graph patterns may be used by combining BGPs to form the query graph pattern of the WHERE clause. Indeed, the query graph pattern may:

- be a union of graph patterns to express alternative graph patterns matching: P1 UNION P2

- have optional graph patterns to allow mandatory patterns results to be possibly extended with additional information by adding the results to the part where the information is available: P1 OPTIONAL P2

- have some restrictions applied to the patterns results: P FILTER (expression)

- have named graph pattern for matching a graph pattern against a specific RDF named graph identified by a variable or URI: GRAPH VARIABLE P or GRAPH URI P

In addition, the SPARQL language provides operators to modify the query solutions:

- DISTINCT: to delete duplicate solutions.

- REDUCED: to enable duplicate results to be eliminated.

- ORDER BY: to order the solutions according to a set of expressions (variables, ASC() and DESC() functions, etc.).

- LIMIT: to give the maximum number of solutions to return.

- OFFSET: to slice the solutions and to indicate the size from which they are generated.

In 2013 new features have been introduced: sub-queries (nested select queries), aggregates (compute expression on a results set), negation (NOT EXISTS and MINUS), select expression (to evaluate an expression and assign its value to a variable), property path (possible route between two nodes of a RDF graph), assignment (BIND and VALUES) and more complex functions and operators.

Furthermore, SPARQL was natively designed for querying remote data sources. Beyond a graph query and update language (to modify RDF data graph), SPARQL defines a protocol for transferring queries towards remote servers (*SPARQL endpoints*) and returning solution sequences.

A SPARQL endpoint is a query processing service based on the SPARQL HTTP protocol which enables to query remote data sources for both machines and humans. Since SPARQL 1.1, the language defines a *SERVICE* clause aiming at applying a sub-query to a specific endpoint identified by its access URI. A SPARQL query can

thus be composed by several SERVICE clauses computed on different endpoints, which results are combined together.

2.4.4 Linked Open Data (LOD) integration

Driven by the 5-stars rating scale, a plethora of triplestores (RDF data sources) were published and connected to the Web. The Linking Open Data project[3] identifies open data sources which are compliant with LOD principles and shows existing links between them. These sources are mainly published by the project members but other organizations and individuals may also participate. The project occasionally publishes a cloud diagram representing the datasets as you can see in Figure 2.4. In ten years (from 2007 to June 2018), the number of datasets increased from 12 to 1224. The published data covers various domains and topics such as life science, music, geography, government or media data. The project also publishes statistics about datasets[4], the vocabularies[5] used within and information [81] about associated SPARQL endpoints. Some notable organizations such as DBpedia [47, 7], BBC, UK Government or the New York Times can be identified among the data providers.

The LOD integration aims at integrating these kind of autonomous and distributed data sources which, as mentioned above, might be considered as a global distributed RDF graph thanks to their RDF links. In order to do so, the LOD integration relies on RDF and SPARQL through the Distributed Query Processing (DQP) approach. Indeed, performing RDF-based data integration is equivalent to virtually integrate homogeneous data sources. Thus, the query answering mechanism is handled by (i) expressing queries (initial and sub-queries) in SPARQL and (ii) sending them to triplestores through their SPARQL endpoint. The vocabularies and ontologies used to describe RDF data represent both global and local schemas. The significant increase of the amount of RDF datasets drives more complex information and queries to deal with. Finding efficient query processing approaches thus became one of the main research question in this area [44] and raised several studies.

[3] http://lod-cloud.net/
[4] http://stats.lod2.eu/
[5] http://lov.okfn.org/dataset/lov/

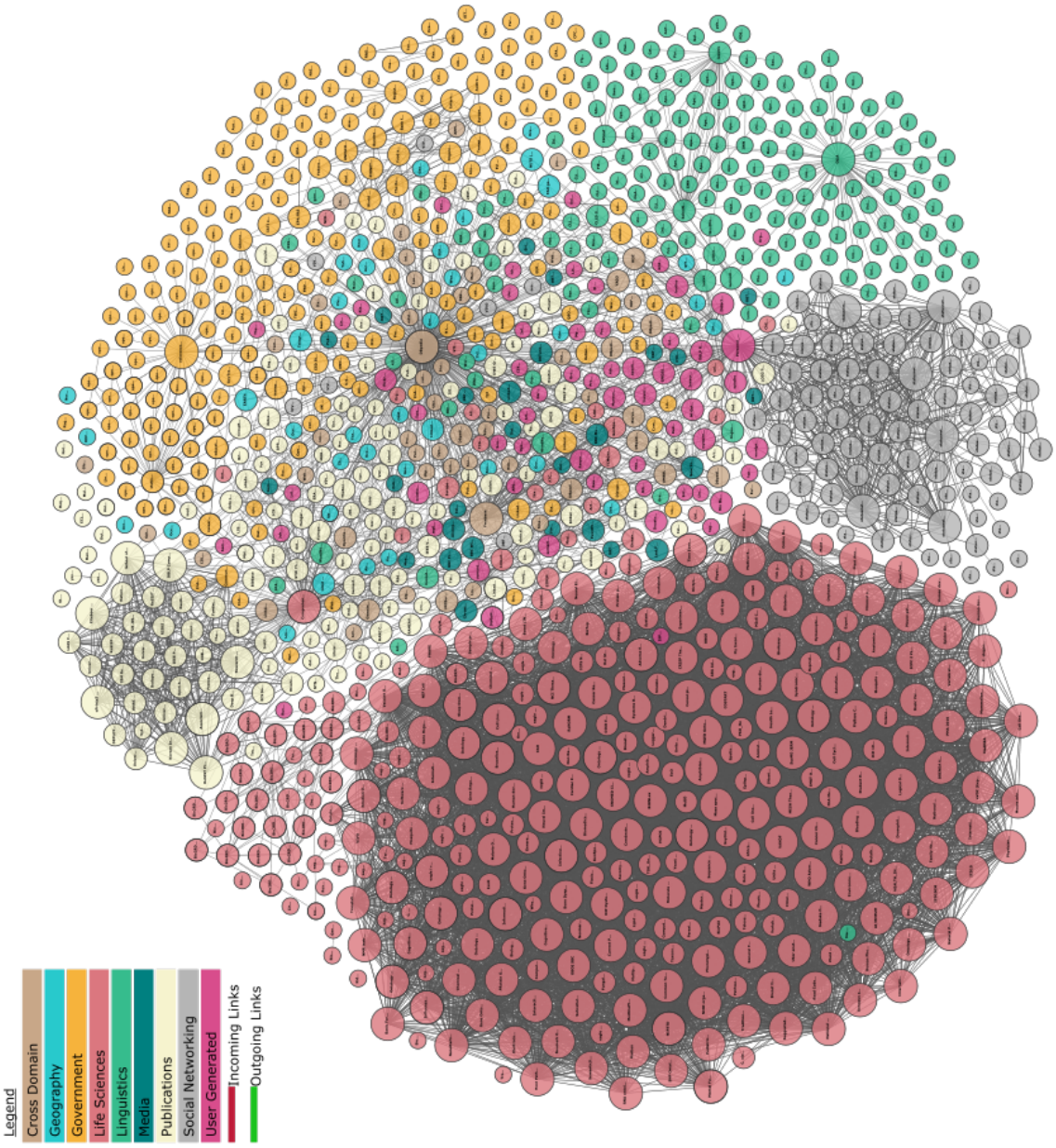


Figure. 2.4: Linking Open Data cloud diagram

3 Federated query processing: State-of-the-Art

Contents

3.1 Introduction **22**
3.2 Distributed Query Processing principle **22**
3.3 Federated SPARQL query processing engines **23**
3.3.1 Source selection 25
3.3.2 Query rewriting 27
3.3.3 Query planning 29
3.3.4 Query evaluation 31
3.4 Discussion and Challenges **33**
3.5 Conclusion **35**

3.1 Introduction

A large number of RDF datasets are made available and are linked by data providers over the Web according to the Linked Open Data principle. SPARQL endpoints are mainly used to provide RDF datasets to the data consumers. The links between these data sources lead to a distributed knowledge graph on the Web. Querying these knowledge bases therefore requires to request them through federated queries.

In this chapter, we review the state of the art for federated query processing over Linked Open Data. We first remind the Distributed Query Processing principles and their main steps (source selection, query rewriting, query planning and query evaluation). Then, in Section 3.3, we introduce several existing SPARQL DQP engines and we describe their approaches for optimizing each step. Finally, we further discuss the described SPARQL federation engines optimization approaches. Meanwhile, we underline the challenges needed to be addressed in order to improve the SPARQL federated query processing performance.

3.2 Distributed Query Processing principle

Given the fact that the knowledge graph is distributed over several data sources, applying the original query to partial data sources is likely to return a subset, possibly empty, of the expected results only since data from different sources need to be joined. A proper DQP strategy will therefore decompose the original query into sub-queries that are relevant for each data source and compute joins later on. Distributed querying typically implies four main steps [58]: source selection, query rewriting, query planning and query evaluation (see in Figure 3.1).

- *Source selection* identifies data sources that are likely to contribute to the result set in order to avoid unnecessary processing. Indeed, all data sources are not necessarily containing data that are relevant for a given query. Thus selecting relevant sources prevents from sending useless remote queries.

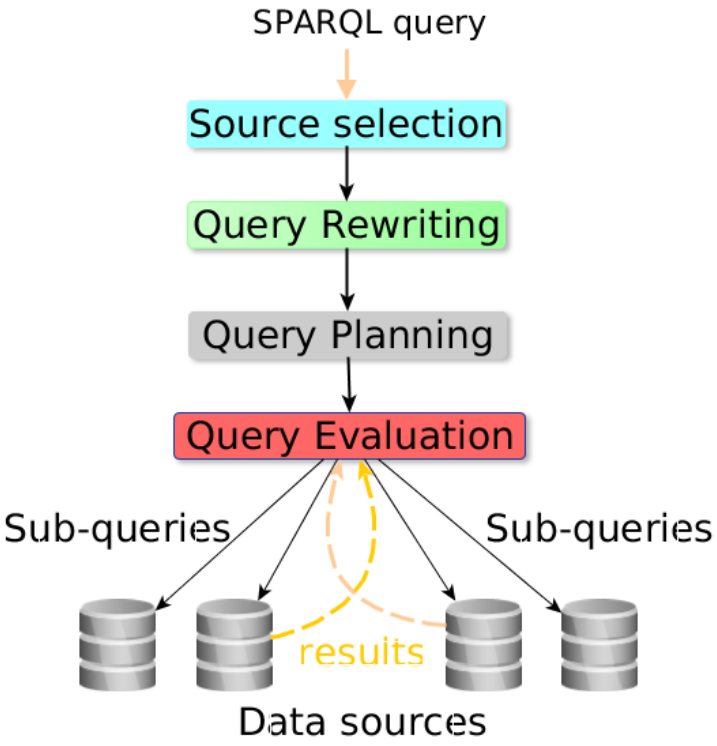
- *The Query rewriting* [16] step decomposes the original query into a set of sub-queries to be distributed to the remote data servers. It both ensures preservation of the original query semantics and takes into account the performance of the sub-queries to be processed.

- The order in which sub-queries are applied can have a drastic impact on performance. *The Query planning* step sorts sub-queries in order to generate an

optimal evaluation plan that minimizes the overall query execution time [63]. As a general rule, the most selective queries should be processed first.

- Finally, the *query evaluation* (or *query execution*) step executes the query plan and collects results from all sub-queries generated. Partial results thus aggregated often require further processing (different result sets typically need to be joined) to compute final results.

Figure. 3.1: SPARQL DQP steps

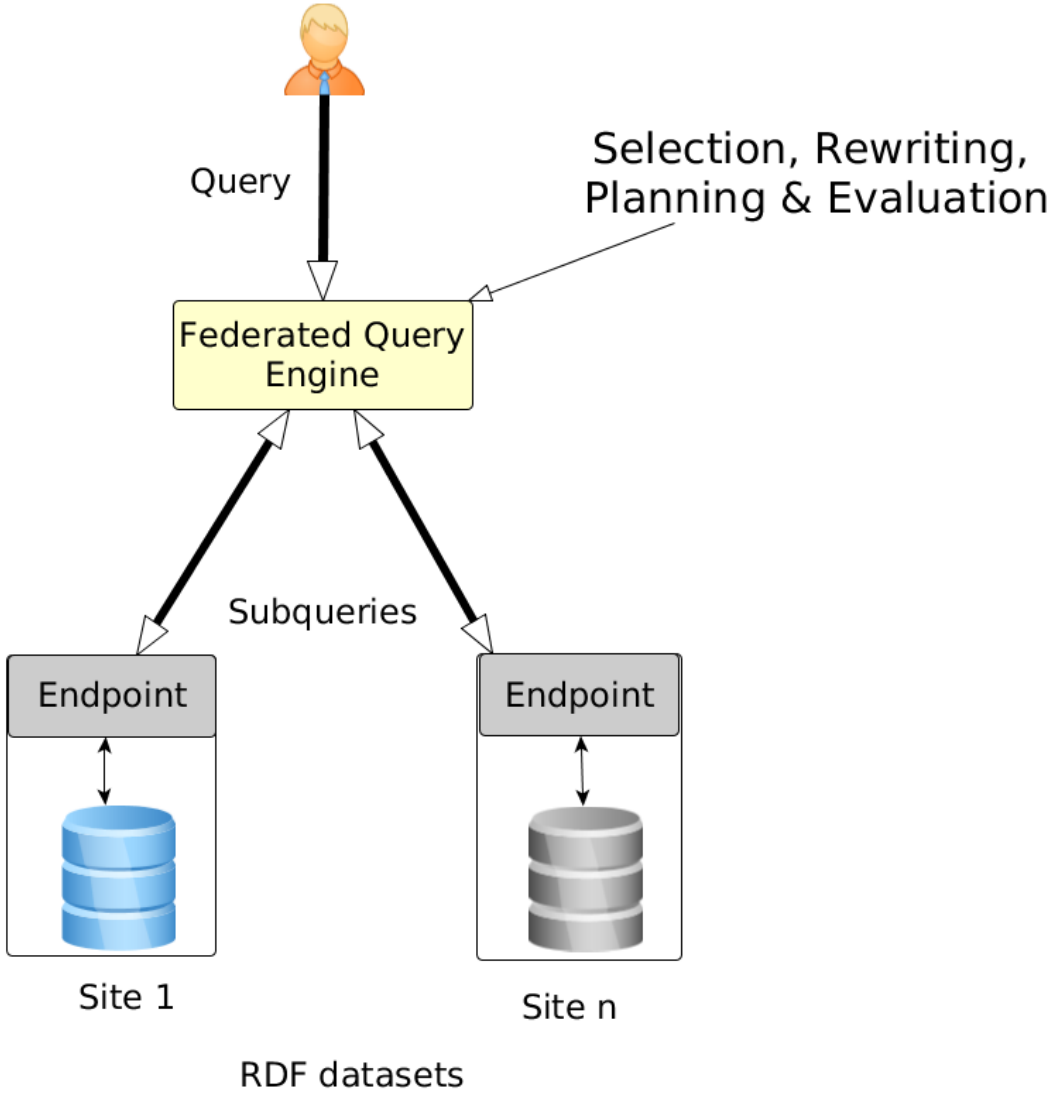


3.3 Federated SPARQL query processing engines

Source selection requires information on the sources content. This information is usually acquired prior to query processing, either through data statistics collection on the endpoints, or by probing the endpoints through source sampling queries. Query rewriting involves decomposing the original query into source-specific sub-queries so as to retrieve all data that contribute to the final result set based on information gathered in the previous step. The query plan orders the sub-queries generated during the query rewriting stage. The evaluation step performs the sub-queries plan execution and incrementally builds the final results set by choosing the most suitable join method and joining intermediate results retrieved from sub-queries.

Naive DQP implementations may lead to a tremendous number of remote queries and generate large partial result sets. The optimization of DQP is therefore critical and many related work primarily address the problem of query performance. This is the case for several DQP engines designed to query federated RDF data stores over SPARQL endpoints (Figure 3.2), such as DARQ [64], FedX [75], SPLENDID [32], ANAPSID [2], LHD [89], WoDQA [4] and ADERIS [51]. Their data source selection, query decomposition strategies, query planning and query execution are described below.

Figure. 3.2: SPARQL DQP engines architecture



3.3.1 Source selection

We have identified two main source selection approaches regarding to the literature: (i) metadata catalogs source selection and (ii) sampling query-based source selection.

3.3.1.1 Metadata catalogs

Metadata catalogs provide descriptions of SPARQL endpoints. These descriptions are usually composed by information on predicates and their related subjects and objects or statistics (cardinalities about number of instances of triples or query patterns, execution time, etc.). Two sorts of metadata catalogs can be distinguished:

- Service Descriptions [80] describes SPARQL endpoints data as statistics on predicates and possible restriction on query patterns.

- VoID [5] descriptions not only provide metadata on SPARQL endpoints data but also indicate their links with other data sources

Based on metadata retrieved, SPARQL DQP engines perform source selection to identify the relevant sources for the initial query. The metadata freshness will impact its accuracy and possible query optimizations for the subsequent steps.

3.3.1.2 Sampling Query-based source selection

Rich prior information on data sources content is useful to implement elaborated source selection strategies. However, it requires specific instrumentation of data sources, and therefore does not apply to most SPARQL endpoints. According to the LODStats [50, 81] project only 13.65% of the endpoints provide a VoID description and 25.18% provide a Source Description. To address source selection for SPARQL endpoints without data description, DQP engines try to build their own data indexing by sending queries to SPARQL endpoints to obtain necessary information. We can classify Sampling Query-based source selection in two groups:

- ASK queries: it consists of sending SPARQL ASK queries to know whether a SPARQL endpoint may provide a result for a triple pattern (TRUE) or not (FALSE). If the ASK query result is TRUE, the data source is relevant to the triple

pattern. In general, ASK queries are used to identify predicates distribution over SPARQL endpoints [42].

- Complex queries: a data source can be relevant for a triple pattern without contributing to the final results. Indeed, intermediate results from one relevant dataset might be later excluded after join processing with results from another relevant dataset. To take this issue into account, some DQP engines use more complex SPARQL queries relying on rdf:type, owl:sameAs, heuristics and joins between triple patterns.

For both metadata catalogs and query-based sampling, caching may avoid to request the same information several times. However caches have to be updated to remain accurate if endpoints data are frequently changing.

3.3.1.3 Federated query engines source selection

Most of existing engines rely on prior information on sources content for source selection. DARQ [64] relies on *service descriptions* to identify relevant sources. The descriptors provide information on predicate capabilities and statistics on triples hosted in each source. DARQ compares service descriptions predicates with the query pattern predicates. For that reason, predicates with variables are not handled.

Similarly, the LHD [89] engine also compares triple pattern predicates and descriptions predicates. It uses VoID descriptions but combines them with ASK queries to refine them. Moreover, unlike DARQ, LHD can support triple patterns with a variable as predicate.

SPLENDID [32] *index manager* uses statistical data on sources expressed through VoID [5] to build an index of predicates and their cardinality. But for triple pattern with predicates not covered by the VoID descriptions, SPARQL ASK queries are used. All sources are assigned to triple patterns with a variable as predicate.

ADERIS [51] source selection relies on list of predicates metadata catalogs. But some SELECT DISTINCT queries on predicates are also sent during execution to update predicates tables. These tables are used for its query evaluation step.

Similar to LHD [89], ANAPSID [2] also has predicate lists catalogs. Besides predicates, ANAPSID catalogs contain predicates and various statistics (e.g. endpoints execution timeout) for all available endpoints. The statistics, also used for query

planning, are updated dynamically by the query engine during the query evaluation step. ASK queries with heuristics are also used to complete the catalogs.

The WoDQA [4] engine selects sources by using VoID information about the dataset links. Based on IRIs analysis, WoDQA selects as relevant source all sources having the query triple pattern IRIs in their links description.

FedX [75] also performs source selection optimization but unlike the previous approaches, without prior knowledge on sources. Only SPARQL ASK queries are sent to identify the relevant sources for querying predicates. Relevant sources are managed through an index and stored in a cache to avoid to repeat the process for the same query.

Table. 3.1: Source selections approaches for SPARQL DQP engines

	Metadata catalogs	Sampling Query-based	Cache
DARQ	Service Descriptions (predicates)		✓
LHD	VoID (predicates)	ASK	
SPLENDID	VoID (statistics)	ASK	
ADERIS		SELECT DISTINCT	
ANAPSID	Predicates list	ASK	
WoDQA	VoID (dataset links and IRIs)	ASK	✓
FedX		ASK	✓

Table 3.1 resumes the SPARQL DQP engines sources solution strategies. Most of them use a hybrid approach by combining both Metadata catalogs to get information on endpoints data and Sampling Query-based either to update and refine the metadata or to obtain further information to perform a more accurate source selection. Only FedX and DARQ are using one approach of source selection. FedX uses the Sampling Query-based approach and DARQ the metadata catalogs one. Both of them are using caches to reduce the communication cost with the SPARQL endpoints.

3.3.2 Query rewriting

Based on the relevant data sources previously identified, the original query is decomposed into source-specific sub-queries so as to retrieve all data that possibly contributes to the final result set. There are mainly two query rewriting methods: Triple-based and BGP-based.

Triple-based evaluation is the finest-grained and simplest query decomposition strategy. It consists in decomposing the query into each of its triple patterns, evaluating each triple pattern on a source independently, collecting the matching triples on the endpoints and performing the intermediate results joins.

The obvious drawback of evaluating triples patterns individually is that many triples may be retrieved from the sources and returned to the federated query engine just to be filtered out by the join operations. If this strategy is returning correct results (all potentially matching triples are necessarily returned), it is often inefficient. Indeed, all the joins processing have to be achieved at the query engine side.

Conversely, the BGP-based strategy consists in evaluating Basic Graph Patterns (BGPs) sub-queries [83]. The BGPs are groups of triple patterns. BGP-based strategy involves grouping the triple patterns in order to reduce the number of remote queries and intermediate results. Indeed, BGPs evaluation processes joins at the endpoint level, which is more efficient than retrieving all possible triples at the server level before computing joins. The use of SPARQL SERVICE clauses is typically a way to send BGPs for evaluation to remote endpoints. However, SERVICE clauses can be used for a whole BGP in a data source only if this data source is relevant for all triple patterns of this BGP. This kind of BGP is also called an *exclusive group* by some federated query engines.

The query rewriting step also depends on the data distribution scheme. Indeed, in the case of vertically partitioned data sources, a whole partition is contained in a single server, to which an *exclusive group* can be sent, in particular through a SERVICE clause. However, in the case of horizontally partitioned data sources, *exclusive groups* can only return a subset of the expected results due to the need to compute joins between data distributed in different data partitions. To preserve the semantics of the original query, it is mandatory to carefully split it into sub-queries that do not cause results losses.

BGP-based query rewriting is used by most of the federated SPARQL query engines to improve performance. DARQ introduces a triple pattern grouping approach and also adds filters pushing to BGP-based sub-queries for more selectivity. This approach was first name a *exclusive grouping* by FedX and reused by SPLENDID. Furthermore, SPLENDID added a heuristic based on triple patterns with owl:sameAs as predicate. When this kind of triple pattern has a variable as subject or object, it is included in the *exclusive group* of the triple pattern with the same variable. This heuristic is based on the assumption that data sources providers only use owl:sameAs predicate to link their own data. ANAPSID also uses a BGP-based approach. WoDQA separates an *exclusive group* into several when some triple patterns do not share variables. SPLENDID and DARQ support SPARQL 1.0, which does not include the SPARQL SERVICE clause. However, in all these approaches, BGP generation is only considered for triples patterns relevant to a single source (vertically partitioned data). For horizontally partitioned data sources, Triple-based query rewriting is used to avoid missing solutions generated by intermediate results from different data partitions.

Table. 3.2: Query Rewriting approaches vs data partitioning

	Vertically partitioned data	Horizontally partitioned data
DARQ	BGP-based (filters)	Triple-based
SPLENDID	BGP-based (owl:sameAs)	Triple-based
ADERIS	BGP-based	Triple-based
ANAPSID	BGP-based	Triple-based
WoDQA	BGP-based (shared variable)	Triple-based
FedX	BGP-based	Triple-based
LHD	Triple-based	Triple-based

Table 3.2 summarizes the SPARQL federated query engines approaches for query rewriting. Both Triple-based and BGP-based evaluations are leveraged by most of the engines with some particularities for the second approach. Only LHD makes use of the Triple-based approach in all cases. However, the BGP-based approach, which is the more efficient one, is only used for vertically partitioned data. Indeed, for horizontally partitioned data, Triple-based evaluation is always used.

3.3.3 Query planning

Besides query rewriting, query planning is another way to optimize a SPARQL query evaluation. Indeed, the same SPARQL query can be rewritten in different statement forms with high processing time variations. A simple sub-queries reordering can significantly reduce the processing time. Indeed, sorting the sub-queries is equivalent to reordering the query joins. Therefore the sub-query sorting is a key aspect of the query planning step of federated query engines.

Dynamic programming [76] is the most commonly used approach to perform query planning. This approach iterates over all possible evaluation plans before selecting one, given a cost estimation function, by progressively pruning the non-optimal ones. *Dynamic programming* involves computing a quick cost estimation function to be efficient. Otherwise it becomes very expensive for complex queries, particularly in terms of processing time since the query planning time is accounted as part of the overall query processing time.

Conversely, the *greedy algorithm* approach generates only one "optimal" plan. This approach starts with the sub-query with the lowest cost and recursively chooses the next one based on a cost estimation function. The plan generated might not be the best one but should be close enough to it to improve the query evaluation within a reasonable time. Thus, the query planning step needs to keep a balance between the accuracy of the plan and query planning time. On the other hand, estimating the optimal query execution plan is known as an intractable problem [45]. Therefore,

the actual purpose of the query planning step is to find an optimal plan close to the best one. To do so, heuristics are more commonly used to define the Cost Estimation functions. In the SPARQL DQP context, heuristics can be classified into two categories: (i) heuristics based on query patterns and (ii) plan cost estimation heuristics

3.3.3.1 Heuristics based on query patterns

Several federated query engines perform query planning by using some heuristics to assess the selectivity of query patterns and sort them based on their selectivity. The main heuristics approaches used are: *Free Variables Counting* [82], *Shared Variables* and *Query Patterns Priority*.

- *The Free Variables Counting (FVC)* heuristic consists in assessing the selectivity cost of a query pattern by counting its number of free variables, assuming that *subject* variables are more selective that *object* variables, which are themselves more selective than *predicate* variables. Then, the query patterns are evaluated according to this number. Free variables which are already bound by a previous query pattern, are also considered as being bound for the following ones.

- *The Shared Variables (SV)* heuristic takes into account the variables reuse between query patterns. Indeed, processing the query patterns sharing variables with a given query pattern previously executed, enables in some cases the engine to propagate the values already known for these variables and potentially reduce their selectivity.

- *The Query Patterns Priority (QPP)* heuristic sorts the sub-queries by giving a priority to some types of query patterns, for instance the *exclusive groups*.

FedX uses both *Free Variables Counting (FVC)* and *Query Patterns Priority (QPP)* heuristics. QPP gives priority to *exclusive groups* over triple patterns. Both of them are sorted through FVC before. In the WoDQA engine, the query planning relies on the *Shared Variable (SV)* heuristic. After re-ordering the query patterns based on a given selectivity coefficient, they are sorted again through the SV approach in order to place alongside query patterns with common variables.

3.3.3.2 Plan cost estimation heuristics

Other federated query engines define a Cost Estimation Function through heuristics in order to perform either dynamic programming or greedy algorithms. These heuristics aim at estimating the joins cardinality, and the communication cost associated to some query patterns. They are defined by considering either statistics on SPARQL endpoints data provided by VOID and Service descriptions, or some observations on RDF features and datasets.

DARQ, LHD, ADERIS and SPLENDID engines use this approach for their query planning. ADERIS performs greedy algorithm, while all others are using dynamic programming. DARQ estimates the cardinality cost with the cardinality retrieved from the service descriptions combined with a selectivity fatcor for each join. Since this factor is missing in the service descriptions, it is arbitrarily fixed to 0.5. Similarly SPLENDID and LHD also estimate the cardinality cost relying on VOID descriptions. However in this case, the selectivity is given by the VOID descriptions.

Table. 3.3: Query Planning approaches for SPARQL DQP engines

	Heuristics on query patterns	Plan cost estimation heuristics
DARQ		dynamic programming
LHD		dynamic programming
SPLENDID		dynamic programming
ADERIS		greedy algorithm
WoDQA	Shared Variables	
FedX	FVC & QPP for exclusive group	

Table 3.3 shows the different query planning approaches for several engines. DARQ, SPLENDID and LHD apply dynamic programming considering a cardinality cost function to generate different plans and chose the best one. But LHD focuses on the optimization of each BGP separately. Conversely, ADERIS, FedX and WoDQA generate one optimal plan. ADERIS uses greedy approach with a cardinality cost estimation, whereas WoDQA and FedX only focus on query patterns selectivity heuristics.

3.3.4 Query evaluation

The first purpose of the *Query Evaluation* step is to send sub-queries to SPARQL endpoints following the plan previously decided and to retrieve intermediate results from remote datasets. These results are then joined in order to produce the final results. The SPARQL query result is called *solution sequence* which is a list of *solution mappings*. A *solution mapping* associates the variables of the query graph pattern to

their matching RDF terms (resources, literals or blank nodes). A solution mapping is a set of *variable bindings* where each variable binding is a pair of a variable name and an associated value). The term *bindings* is commonly used to refer to query variable bindings and, more generally, to already known values for variables.

To compute joins, several kind of operators may be used:

- nested-loop join: processes a join between two relations in a nested loops fashion. It iterates over the bindings of the first relation for each binding of the second relation.

- bind join [34]: unlike nested-loop, bind join propagates the bindings of the first relation to the second relation instead of iterating over the results and propagating one by one known values.

- merge join: also called sort merge join, aims at merging two sorted relation bindings.

- hash join: builds a hash table of the smallest relation bindings and compares them with the larger relation bindings. The latter, are also hashed before checking if they match with hash table values.

The join operations aggregate intermediate results from different remote data sources. The DARQ engine implements both nested-loop join (NLP) and bind join (BJ). NLP is used for accessing patterns without limitations and BJ in the opposite case. SPLENDID has two query evaluation strategies: bind join evaluation and parallel evaluation. On the one hand, it aims at identifying independent sub-queries, sending them in parallel and using hash join (HJ) to locally aggregate the results. On the other hand, bind join is used to sequentially evaluate dependent sub-queries. Similarly, LHD makes use of both bind join and hash join operators respectively.

The FedX engine query evaluation relies on bind joins and SPARQL UNION queries to send all the mappings in one remote query to data sources. FedX also performs a parallel evaluation trough a multithreading approach and a pipelining strategy to process intermediate results as early as possible. Bind joins are also used by the WoDQA engine to execute queries.

The ADERIS engine query evaluation builds predicate tables for each sub-query to complete subjects and objects values. The join operation through nested loop join starts once the tables are complete. This approach is called index nested loop join. Indeed, the tables values are used as index on join attributes.

Finally, ANAPSID [2] is an adaptive query engine since its query evaluation is based on two types of joins: adaptive group join (AGJ) and adaptive dependent join (ADJ). AGJ relies on Symmetric Hash Join [23] and Xjoin [85]. AGJ detects when a data source becomes blocked and adapts the query evaluation according to the data availability. ADJ extends a Dependent join operator [26] and sends a query once the results of the dependent query are retrieved from the first data source. Therefore, ANAPSID is capable of quickly producing the first final results.

Table. 3.4: Query Evaluation approaches for SPARQL DQP engines

	Join methods	Parallelism
DARQ	Nested Loop Join & Bind Join	
LHD	Hash Join & Bind Join	Hash join in parallel
SPLENDID	Hash Join & Bind Join	Hash join in parallel
ADERIS	Index-based Nested Loop Join	
ANAPSID	AGJ & ADJ	Join in parallel
WoDQA	Bind Join	
FedX	Bind Join	Multithreading & join pipelined

Table 3.4 sums up the different join methods used by SPARQL federated query engines for query evaluation and their parallelism approach. Different kind of joins are performed by the engines based on their query evaluation strategy. Bind join is the most frequently used operator in order to propagate bindings to the followings sub-queries and therefore to reduce intermediate results. Bind join is combined with Nested Loop join or Hash join operators in DARQ, LHD and SPLENDID. While FedX defines an optimized version of bind join with SPARQL UNION queries to reduce remote queries. ADERIS uses an index-based nested loop join. ANAPSID defines two joins operators adaptive group join (AGJ) and adaptive dependent join (ADJ) which tailor its query evaluation strategy at runtime and quickly produce the first results. Some parallelism mechanisms are also operated to improve the query evaluation efficiency. In that respect, LHD and SPLENDID achieve hash joins in parallel as well as ANAPSID for AGJ and ADJ join operators. Lastly, FedX completes joins operation in a pipelined fashion and also uses multithreading for its query evaluation approach.

3.4 Discussion and Challenges

In the previous section, we compared several SPARQL federated query engines approaches through the distributed query processing main steps. The primary purpose of these engines is to improve the efficiency of SPARQL federated query processing. To achieve this goal, several optimization techniques are implemented. These optimization can be categorized as static and dynamic. Static optimization

refers to optimization processed before the effective query evaluation, whereas dynamic optimization refers to those done during the query execution phase. In particular, the optimization strategies aims at reducing the processing time and the communication cost between the query engine and remote data sources.

The first issue to deal with is identifying the relevant data sources with accuracy. Indeed, overestimating the data sources tends to lead to useless communications and intermediate results processing and consequently to increase the overall distributed query processing time. Conversely, underestimating the relevant data sources will prevent engines from retrieving all expected results. As we have seen in Section 3.3.2, Metadata catalogs and Sampling Query-based are the two main approaches for the source selection step. The first approach, used by DARQ for instance, relies on a prior knowledge (e.g. predicates, statistics, links) on data sources trough VOID and Service Descriptions provided by the endpoints. The efficiency and accuracy of this approach depend on the catalogs freshness. Therefore, the Metadata catalogs need to be frequently updated to preserve the results accuracy and completeness. However, data catalogs are expensive to keep up to date [84].

We have also noticed that very few endpoints provide these catalogs, which forces federated query engines to use the Query-based Sampling approach to try to achieve accurate source selection. This method, performed by the FedX engine, consists in sending sampling queries to endpoints to retrieve information and in building its own data catalog. This approach will more likely provides up-to-date results. The main drawback of the Query-based Sampling method is the unavoidable processing time for retrieving the information and for building the index on the fly.

As seen in the previous section, most of the SPARQL federated query processing such as LHD, SPLENDID, ANAPSID and WoDQA perform a hybrid approach by mixing the two foregoing approaches. The former provides default information on data sources and the latter completes, updates or refines it.

Regarding the query rewriting optimization, all the engines try to take advantage of the BGP-based approach, which is more efficient than the Triple-based approach. However, its use is restricted to *exclusive groups* and therefore to vertical partitions. Indeed, a naive usage of the BGP-based approach for horizontal partitions may lead to miss results generated by different data sources. The question then arises of how to benefit from the advantage of the BGP-based method in horizontally partitioned data without losing accuracy or completeness. This question is related to the more general issue of query semantics preservation in context of distributed data sources. Indeed, the query rewriting step has to both take into account the performance of the sub-queries generated and to ensure the preservation of the original query

semantics. The SPARQL distributed query semantics is not addressed by the engines reviewed in this chapter.

Data replication in RDF data sources is another challenge that needs to be taken into consideration by the federated query engines. Data redundancy is very common in the Linked Open Data context since several data sources are built by crawling other sources. Consequently, data replication may have a negative impact on the query processing performance by increasing the number of intermediate results, increasing the communication cost, and by providing duplicated results. Thus, data replication should be taken into account, either by detecting it during the source selection step (in order to avoid to resend a sub-query which will retrieve intermediate results already provided by another source), or during the query evaluation step. None of the reviewed engines investigate this issue. However, DAW [72], built on the top of FedX, and BBQ [43] propose a duplicate detection mechanism throughout the source selection phase to estimate data overlapping among data sources relying on data summaries. Fedra [54] takes into account the fragment replication in data sources during source selection step to reduce the size of data transferred.

Regarding the SPARQL language expressiveness, only ANAPSID and WoDQA can cope with some SPARQL 1.1 features. The initial version of FedX, SPLENDID, LHD and DARQ support SPARQL 1.0. However, FedX has been updated to support several SPARQL 1.1 features. ADERIS supports a subset of SPARQL 1.0 query patterns and can not handle features such as OPTIONAL. SPARQL expressiveness, emphasized with SPARQL 1.1, should be preserved by the federated query engines as much as possible. The increase number of RDF data sources creates more complex information to query. Consequently preserving the expressiveness is essential to avoid users to be limited in their data querying capability.

3.5 Conclusion

In this chapter, we have first described the Distributed Query Processing (DQP) approach which is the main method used for Linked Open Data integration. Integrating distributed and autonomous RDF data sources is equivalent to performing a virtual integration of homogeneous data sources. Applying the DQP approach requires achieving the source selection, query rewriting, query planning and query evaluation steps. Afterward, we have described several current SPARQL federated query engines approaches to address these stages. For each step, we have compared the different optimizations performed by these engines. Finally, we have discussed in detail the approaches proposed to improve the federated query processing efficiency related to the processing time and the communication cost. We also have

highlighted the challenges to tackle such as the source selection accuracy, the results completeness, the data replication, the SPARQL distributed query semantics and its expressiveness.

The first aim of this thesis is to propose a clear SPARQL and RDF-compliant Distributed Query Processing semantics in order to ensure the query semantics and expressiveness preservation. The following goal is to develop a SPARQL federated query engine that transparently addresses distributed data sources, while taking into account the previously underlined challenges.

In this thesis, we propose a federated query semantics taking into account distributed RDF datasets constraints and specify the semantics of federated SPARQL queries on top of standard SPARQL semantics to respectively enhance their reliability and expressiveness. Afterwards, we propose an approach to transparently query distributed data sources while addressing data partitioning, data replication and query results completeness challenges through both static and dynamic optimization strategies.

4 Towards Federated SPARQL query semantics and specifications

Contents

4.1 Introduction 38
4.2 SPARQL query evaluation over a distributed knowledge graph . . 38
4.3 Federated dataset semantics on top of RDF semantics 42
4.4 SPARQL features over distributed data sources 43
4.4.1 Rewrite rules 44
4.4.2 Triples replication 48
4.5 Conclusion 49

4.1 Introduction

SPARQL federated query engines aim at virtually integrating several autonomous and distributed RDF data sources through their endpoints. This federation should be transparent from the query designer point of view as if all sources were aggregated in a single data source. As mentioned in the previous chapter, dedicated engines perform the distributed queries evaluation over a *Virtual Knowledge Graph* (VKG) aggregating all source RDF graphs. Owing to the knowledge graph distribution, an initial query needs to be rewritten into sub-queries in order to interrogate each source on its own data before performing joins. Otherwise, applying the whole query over remote sources is likely to retrieve partial or empty results. Therefore, the distributed query semantics preservation becomes a crucial question to address. If the semantics of SPARQL queries is clearly defined for a single data source[1], this is not the case for federated SPARQL queries. The assumption intuitively made in the current state of the art is that a SPARQL query evaluation over multiple data sources should return the same results as if all data sources had been managed as a single one and the *VKG* had been materialized. However, as we are going to see further in the next section, making the federated query engines compliant with this assumption is not straightforward.

In this chapter, we first investigate federated SPARQL query semantics issue through a comparison between a standard SPARQL query execution over a single data source (centralized RDF dataset) and a federated SPARQL query execution over several data sources (distributed RDF dataset). Then, we express the constraints that federated engines should comply with. In a second step, we express federated SPARQL queries through the standard SPARQL features in order to define their semantics on top of standard SPARQL semantics.

4.2 SPARQL query evaluation over a distributed knowledge graph

Semantic Web knowledge and information are represented through the RDF data model. In that respect, the Linked Data query federation data sources are RDF datasets. Therefore the standard SPARQL semantics is defined according to the RDF semantics [67]. As defined in this semantics, an RDF dataset[2] is a set of RDF graphs and is composed of two main elements: one default graph and zero or more named graphs. The default graph may be empty and graph names are unique in a

[1] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#sparqlAlgebraEval
[2] https://www.w3.org/TR/rdf11-concepts/#section-dataset

given RDF dataset. The RDF dataset graphs can share blank nodes between them. RDF specification does not allow triples replication in an RDF graph, even though triples may be replicated in different RDF datasets either to improve their availability. Standard SPARQL queries are evaluated over RDF datasets in accordance with this semantics [67].

Federated SPARQL queries are evaluated assuming that data sources are a distributed RDF dataset, or *Virtual Knowledge Graph (VKG)*, aggregating all RDF datasets. Since the data distribution should be transparent to users, most works on federated SPARQL engines consider that users expect the SPARQL query to be evaluated in the "union" of the RDF datasets. This "union" would correspond to a single RDF dataset with the union of the default graphs as default graph and the union of the named graphs as named graphs.

Intuitively, it also means that the VKG, on which federated queries are evaluated, should be compliant with RDF semantics. Nevertheless, implementing this semantics is not obvious. Indeed, to define a semantics for federated SPARQL queries evaluation, there are RDF graph particularities such as *named graphs*, *blank nodes* and *data replication*, and the impact of the latter on SPARQL *redundant results* handling need to be considered.

This semantics specification has to deal with the constraints below:

- Named graphs are graphs in an RDF dataset, identified by an IRI. In an RDF dataset, the graph names are unique. So what is the relation between two named graphs in different RDF datasets with the same IRI? According to the RDF standard, the fact that the same IRI for named graphs appears in different RDF datasets does not imply anything on the relation between these named graphs. Thus, there is an alternative in dealing with this problem since there is no semantics for the relation between the named graph and its URI:

  - the named graphs can arbitrarily be considered as different: the SPARQL query would be separately evaluated in the two graphs

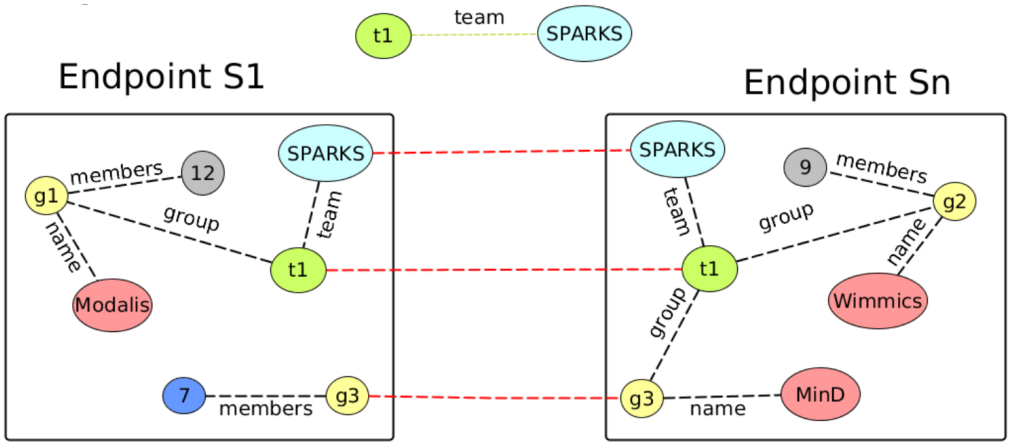
  - the named graphs can arbitrarily be considered as same named graph : the SPARQL query would be evaluated in the merge of the two named graphs.

- Blank nodes are locally scoped nodes which identifier, if any, is not a IRI. Hence identifiers in SPARQL results of two blank nodes from two different endpoints may collide. In a single dataset, graphs can share blank nodes (for instance

two named graphs). However, in the federated context blank nodes identifiers collisions in SPARQL results may occur for unrelated nodes from two data sources or a given blank node may not have the same identifier in two SPARQL results. Therefore, the blank nodes distributed join processing is not always feasible or may result in inappropriate results.

- Data replication: triple redundancy in a single RDF graph is not allowed by RDF semantics, but in a distributed context, it is common to replicate data items over several data servers to improve availability [58]. RDF triples may thus be replicated in different servers.

Figure. 4.1: Data replication on Virtual Knowledge Graph



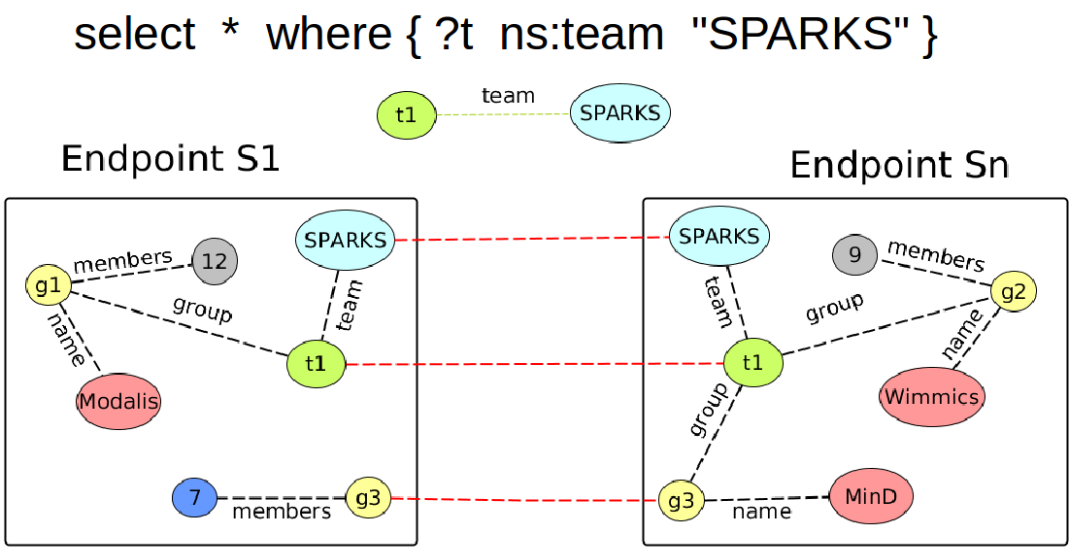
In Figure 4.1, the triple *(t1 team "SPARKS")* is unique in RDF datasets S_1 and S_n, but replicated in the VKG which is the collection (S_1, S_n).

- Redundant results: SPARQL can return some results multiple times in a result multiset, if a graph pattern matches several times the same data. In a distributed context, data replication is also likely to lead to redundant results if the same triple is accounted for several times in a query results. However, this redundancy in results is only a side effect of data duplication that would disappear if data was effectively aggregated in a single RDF dataset.

The Figure 4.2 shows a SPARQL query executed over the VKG (S_1, S_n). The number of results is 2 whereas it would be 1 if S_1 and S_n were actually merged in a single RDF graph.

In the remainder of this thesis, we will refer to Virtual Knowledge Graph and to Federated dataset as follows:

Figure. 4.2: Side effect of data replication



Definitions:
Let $S_1, S_2, ...S_n$ be a set of RDF datasets where S_i is a dataset with d_{G_i} as default graph and $(u_{i_1}, G_{i_1}), (u_{i_2}, G_{i_2}), ...(u_{i_k}, G_{i_k})$ as named graphs.

Definition 4.2.1. *The Virtual Knowledge Graph (VKG) is a collection of the RDF datasets of the remote data sources: VKG* $= (S_1, S_2, ..., S_n)$.

Definition 4.2.2. *The Federated dataset (FDS) is the* ***RDF Dataset Merge*** *of* $S_1, S_2, ...$ *and* S_n *as defined in the SPARQL semantics [79].*

According to this semantics the *RDF Dataset Merge* of two RDF Datasets is defined as follows:

Let $S_1 = \{d_{G_1}, (u_{1_1}, G_{1_1}), ..., (u_{1_n}, G_{1_n})\}$, $S_2 = \{d_{G_2}, (u_{2_1}, G_{2_1}), ..., (u_{2_m}, G_{2_n})\}$ and S the RDF Dataset Merge of S_1 and S_2.

S $= \{d_G, (u_1, G_1), ..., (u_k, G_k)\}$

where:

- d_G is the merge of the default graphs d_{G_1} and d_{G_2}.
- (u_i, G_i) where G_i is from S_1 if u_i belongs only to named graph URIs of S_1.
- (u_i, G_i) where G_i is from S_2 if u_i belongs only to named graph URIs of S_2.

- (u_i, G_i) where G_i is the merge of G_{1_j} and G_{2_k} if G_{1_j} and G_{2_k} respectively belong to S_1 and S_2 and share the same URI u_i.

Based on this definition, we define the Federated dataset (FDS) of $S_1, S_2, ... \, and \, S_n$ as follows:
FDS($S_1, S_2, ..., S_n$) = $\{d_{G_{FDS}}, (u_{FDS_k}, G_{FDS_k})\}$
where:

- $d_{G_{FDS}}$ is the merge of the default graphs d_{G_i} of the datasets S_i $(1 \leq i \leq n)$.
- (u_{FDS_j}, G_{FDS_j}) where G_{FDS_j} is from S_m if u_{FDS_j} belongs only to named graph URIs of S_m.
- (u_{FDS_j}, G_{FDS_j}) where G_{FDS_j} is the merge of named graphs from S_i $(1 \leq i \leq n)$ and sharing the same URI u_{FDS_j}.

Triple replication may occur in *VKG* as shown in Figure 4.2. However, in the *FDS* this replication disappears as result of the merge operation. Thus, for instance in Figure 4.2 the *FDS* for S_1 and S_n would give 1 result to the SPARQL query.

4.3 Federated dataset semantics on top of RDF semantics

The main existing federated engines focus on improving the query processing performance with regard to the execution time and communication cost. This performance is mainly measured through parameters such as source selection time, evaluation time, number of intermediate results and result completeness. The constraints listed above are most of the time disregarded while they may have a significant impact on both the number of distinct results retrieved and their redundancy. The number of results can vary because of the the data partitioning and duplication effect [53]. In a similar way, the processing of named graphs distribution and blank nodes collision may also change the query evaluation results. The disparity between the results provided by the SPARQL federated query engines, for the same query over the same endpoints, shows their lack of reliability. Consequently, it is essential to define a clear semantics for federated queries evaluation and to make it as compliant as possible with both standard SPARQL and RDF semantics.

Hence, we propose the approach below to address the constraints identified beforehand in our SPARQL federated query evaluation strategy:

- Named Graph collision: in the federated context, we consider that data associated to the same named graph IRI belong to a single named graph, even if it is distributed in different sources. This is coherent with the idea of aggregating all data sources into a single virtual graph. In addition, IRI are usually specific enough to prevent most accidental collisions between identical names.

- Blank Node collision: it is not possible to properly address the blank nodes distributed joins issue with their identifiers that are exchanged through query results. To accurately achieve this distributed joins, IRIs need be assigned to these blank nodes before RDF data duplication or distribution. This process is called skolemization[3]. Thus, we advocate the use of skolemization to overcome the blank nodes distributed joins problem and, failing that, to consider the blank nodes identifiers between two given results sets always differ. In the remainder of this thesis, we make this hypothesis.

- Triple replication: when evaluating a query, an RDF triple should be accounted for only one time, even if it is replicated over different servers. In other words, federated queries should be evaluated as if they were processed over the federated dataset defined above which does not contain duplicated triples.

- Result redundancy: the query engine needs to detect which redundant results are legitimate (products of the SPARQL evaluation) and which ones are side effects and should be filtered out.

4.4 SPARQL features over distributed data sources

On one hand, we aim at proposing a SPARQL and RDF-compliant federated query semantics and preserving SPARQL expressiveness as much as possible in the distributed context. On the other hand, the SPARQL service clause is the only semantics[4] clearly defined for federated query and endpoints are mainly using standard SPARQL interpreters. Therefore to specify the federated SPARQL query semantics in compliance with the standard SPARQL semantics, we propose to rewrite SPARQL federated queries into standard SPARQL queries with service clauses.

In practice, processing a federated SPARQL query means executing a SPARQL query over several data sources through their SPARQL endpoints. Thus, given a set of SPARQL endpoints URIs a query pattern has to be evaluated on each relevant SPARQL

[3] https://www.w3.org/TR/rdf11-concepts/#section-skolemization
[4] https://www.w3.org/TR/sparql11-federated-query/#fedSemantics

endpoint. For each triple pattern the union of results provided by endpoints has to be computed and joined with the results of the other triple patterns.

4.4.1 Rewrite rules

A federated query is a SPARQL query provided with a set of SPARQL endpoint URIs. As general rule, a federated query EXP is evaluated as the joins of the results of each triple pattern of EXP, each result being obtained through the union of several service clauses with the same triple pattern and different service URI.

Definition:

Let S = $\{S_1, S_2, ...S_n\}$ be the endpoint URIs and EXP = $\{T_1, T_2, ..., T_k\}$ be a set of triple patterns.

service(S) {EXP} ::= join { $R_{T_1}, R_{T_2}, ..., R_{T_k}$ } where

R_{T_i} ::= $\cup_{s \in S}$ { service s $\{T_i\}$ }

::= {service S_1 $\{T_i\}$} union {service S_2 $\{T_i\}$} union ... {service S_n $\{T_i\}$}

Below, we propose federated SPARQL query rewrite rules for each SPARQL statement:

- **Triple Pattern**: a triple pattern is rewritten as a union of service clauses. Since for a federated query, it is recommended to evaluate expression only on relevant data sources, we introduce the function *Relevant*. This function provides the relevant data sources for each triple pattern. Two cases can be distinguished: (i) query without from clause, and (ii) query with from clause.

  let S = $\{S_1, S_2, ...S_n\}$ and Triple pattern T to evaluate over S.

  Relevant (S, T)::= ST where ST is subset of S containing only the relevant data sources for T.

  Without from clause:

  rewrite(triple T) ::= service(ST) { T }

  ::= $\cup_{s \in ST}$ $\{service\ s\ \{T\}\}$

With from clause: For a query with *from* clause, the ideal rewrite rule would be:

Let F = $\{f_1, f_2, ..., f_n\}$ be a set of URIs of the *from* clause and **from F** be the shortcut for: **from** f_1**, ..., from** f_n.

rewrite(triple T) ::= service(ST) { select * from F where { T } }

However, SPARQL does not allow to specify a dataset in a sub-query, therefore the query above is illegal. But since only one triple pattern is evaluated, achieving this over a *From* f_i can be approximated by processing it over a *graph* f_i under two conditions. Indeed, a From clause implies the merge of the named graphs without duplicates and with blank nodes renaming. The blank node renaming condition is handled through the skolemization assumption but that related to the duplicates need to be taken into account in the rewrite rule. We achieve this by adding a *DISTINCT* clause to the rewrite rule. Thus, we relying on named graph, we propose the rewrite rule below:

rewrite(triple T) ::= select distinct * where {

service(ST) { graph f_1 { T } } union ... union service(ST) { graph f_n { T } }}

- **Basic Graph Pattern (BGP)**: each triple pattern of the BGP is rewritten.

  rewrite(BGP $\{T_1, T_2, ...T_n\}$) ::= BGP {rewrite(T_1), $rewrite(T_2)$, $...rewrite(T_n)$}

- **Union**: each graph pattern is rewritten

  rewrite(P_1 union P_2) ::= rewrite(P_1) union rewrite(P_2)

- **Minus**: each graph pattern is rewritten

  rewrite(P_1 minus P_2) ::= rewrite(P_1) minus rewrite(P_2)

- **Optional**: each graph pattern is rewritten

  rewrite(P_1 optional P_2) ::= rewrite(P_1) optional rewrite(P_2)

- **Filter Expression**: is recursively rewritten and remains unchanged, except for *exists* pattern which is rewritten as a BGP. Thus, each triple pattern of the *exists* expression is rewritten.

- Filter (F) :
  rewrite(Filter F) ::= Filter rewrite(F)

- CST :
  rewrite(CST) ::= CST

- VAR :
  rewrite(VAR) ::= VAR

- $F(EXP_1, ..., EXP_n)$:
  rewrite($F(EXP_1, ..., EXP_n)$) ::= $F(rewrite(EXP_1), ..., rewrite(EXP_n))$

- Exists:
  rewrite (exists {EXP})::= exists {rewrite(EXP)}

  Furthermore, the same rewriting is also performed in all other expressions such as *select, bind, group by, order by and having*, in which *exists* pattern may appear.

- **Sub-query**: the inner query is rewritten using the same rules as for the outer query.

  rewrite(select PROJECTION where{EXP}) ::= select rewrite(PROJECTION) where {rewrite(EXP)}

  For outer query with named graph, the named graph has to be propagated for the sub-query (inner query) evaluation (e.g. using overload rewrite(triple T)).

- **Named Graph Pattern**: named graph pattern is rewritten by overloading rewrite(triple T). There are two cases to consider, depending on the identification of the named graph by a URI or a variable:

  Let G be the federated dataset URIs, which means the set of URI of named graphs in remote SPARQL endpoints.

  - Graph URI:

    rewrite(graph URI EXP) ::= rewrite(EXP)

    overload rewrite(triple T) ::= service(S) { graph URI {T} }

– Graph VAR:

rewrite(graph VAR EXP) ::= union(g in G) { rewrite(EXP) values VAR {g} }

And for each g: overload rewrite(triple T) ::= service(S) { graph g {T} }

In the first case, the named graph is identified by a URI which is propagated into the *service* clauses to focus graph matching.

In the second case the named graphs are identified by a set of URIs in G. The aim of this rewrite rule is to replace the variable of the named graph pattern by a set of Graph URI clauses.

The reason is that SPARQL semantics requires that the BGP of the named graph pattern must be evaluated without the binding of the named graph pattern variable and that the value of the variable of the named graph pattern must be joined to the solution afterward.

The point is that, in some cases as shown in the example below (Query 4.1), the same variable used for the named graph is also used inside the graph pattern. For instance, in this example the variable ?g is both used as the named graph variable and in the *minus* clause. In such cases, the variable in the *minus* must not be bound to the name of the graph pattern.

Query Example 4.1: named graph and minus sharing variable

```
1   select * where {
2    graph ?g { ?x us:p1 ?y minus { ?y us:p2 ?g } }
}
```

- **Property Path**: Regarding the property path, we rely on the property path evaluation defined by the SPARQL recommendation[5]. Indeed, we recursively follow the property path evaluation semantics defined in the recommendation until we reach a triple pattern (X IRI Y). Then we rewrite its evaluation as a service clause. Property path (PP) expressions are made of predicates and operators. The PP interpreter recursively evaluates complex PP expressions with operators that eventually resume to predicates (e.g. p1 / p2). In the context of federated query, evaluating a predicate in a property path expression consists in evaluating the result of the triple pattern rewrite rule with the predicate, the subject and the object as well as the reverse, negation, alternative

[5]https://www.w3.org/TR/sparql11-query/#sparqlPropertyPaths

and sequence operators. Evaluating a predicate consists of finding triples that match the predicate and whose subject (resp. object) match current subject (resp. object). There are several cases: subject and object being known or unknown which respectively means being a RDF term or a variable.

Below we define the rewrite rule for the different property path patterns:

- Predicate Property Path: (S p O) S and P may be RDF Term or Variable
  1. case: (S:var p O:var) ::= rewrite(?s p ?o)
  2. case: (s p O:var) ::= rewrite(s p ?o)
  3. case: (S:var p o) ::= rewrite(?s p o)
  4. case: (s p o) ::= rewrite(s p o)
- ZeroOrMorePath (*), OneOrMorePath (+) and ZeroOrOnePath: in the semantics, these operators are evaluated through functions which take as parameter a path(X,P,Y). This path is rewritten in the same way as in the previous rule.
- Inverse Property Path : ^(S p O) ::= rewrite(O p S)
- Alternative Property Path : (S $p_1|p_2$ O) ::= rewrite(S p_1 O) union rewrite(S p_2 O)
- Sequence Property Path : (S p_1/p_2 O) ::= BGP (rewrite(S p_1 X), rewrite(X p_2 O))

  Sequence, as well as Inverse and Alternative, arguments may be property path expressions, in this case the arguments a recursively rewritten.
- NegatedPropertySet : (S !($p_1|p_2|...|p_n$) O) ::= BGP(rewrite(S ?p O), filter(?p not in ($p_1...p_n$))

4.4.2 Triples replication

To be compliant with the semantics defined above, the possible redundancy in results which are side effects of data distribution should be pruned. In the case of triple pattern it means avoiding redundancy in the results. However, this is not performed

by the standard SPARQL union operator used in the triple pattern rewrite rule. To overcome this issue, we propose to use the *DISTINCT* solution modifier to remove results redundancy by applying a *SELECT DISTINCT* * on the union of the services clauses as follows:

rewrite (triple T) ::= select distinct * where {

{service S_1 {T} } union {service S_2 {T}} union ... {service S_n {T}}

}

In other words, we eliminate several occurrences of the same triple in different endpoints.

However, regarding federated queries with *FROM NAMED* clauses, the results redundancy pruning must be respectively applied only on each named graph evaluation and triple pattern evaluation.

Query Example 4.2: from named rewrite rule

```
Let G = {g1, g2}
select from named g in G where {graph ?g EXP} ::=
{select * where {
        rewrite(graph g1 EXP)}
        values ?g {g1}
}
union
{select * where {
        rewrite(graph g2 EXP)}
        values ?g {g2}
}
The triple pattern rewrite rule (rewrite(triple T)) is
overloaded above for the evaluation of each named graph
g in G with the SELECT DISTINCT * statement.
```

4.5 Conclusion

In this chapter we have shown the necessity to define a clear semantics for federated SPARQL queries in order to ensure the reliability of federated query results and to avoid the disparity between the results they provide. However, this semantics has to

face some constraints related, on the one hand to some RDF graph specific features such as named graph and blank nodes, and on the other hand to data replication and redundant results of RDF distributed graphs. Thus, we propose a federated query semantics while addressing these constraints: (i) named graph collision, (ii) blank nodes, (iii) triple replication.

Afterwards, we specified the semantics of federated SPARQL queries on top of the standard SPARQL semantics by a set of rewrite rules using the service clause since it is the only statement with a clearly defined federated semantics. As a general rule for defining their semantics, federated queries are rewritten as a union of service clauses targeting the remote SPARQL endpoints.

5 Towards efficient Federated SPARQL Query Processing

Contents

5.1 Introduction **52**
5.2 Hybrid sources selection **52**
5.2.1 Sample source selection 53
5.2.2 Source selection algorithm 55
5.3 Hybrid query rewriting **57**
5.3.1 Global join decomposition 61
5.3.2 Joins generation strategy 61
5.3.3 Query rewriting algorithm 63
5.3.4 Hybrid rewriting expressions evaluation 69
5.4 Sub-queries sorting optimization **72**
5.4.1 Cost estimation algorithm 72
5.4.2 Cost-based and shared links sorting approach 74
5.5 Query evaluation **76**
5.5.1 Bindings strategy 76
5.5.2 Parallelism strategy 76
5.6 Conclusion **77**

5.1 Introduction

In the previous chapter, we have defined a semantics for federated SPARQL queries and specified it on top of standard SPARQL. Now, we aim at implementing a more expressive and efficient federated query processing engine in compliance with this semantics while taking into consideration the data distribution, data replication and results completeness challenges. As explained beforehand, SPARQL federated query engines intend to transparently integrate distributed and autonomous RDF data sources through their endpoints. The query processing is mainly performed with the following steps: (i) source selection, (ii) query rewriting, (iii) query planning and (iv) query evaluation.

In Chapter 3, we have seen that several engines, such as DARQ [64], FedX [75], SPLENDID [32], ANAPSID [2], LHD [89], WoDQA [4] and ADERIS [51], have been proposed. As a reminder, most of them use hybrid sources selection approach by combining metadata catalogs and sampling query-based approaches to get information on sources in order to accurately identify the relevant ones. Regarding the query rewriting step, Triple-based and BGP-based evaluations are respectively used for vertically and horizontally partitioned data. There are two main approaches for the query planning step, both based on heuristics. The first one uses heuristics with search algorithms (dynamic and greedy) and the last one applies heuristics on query patterns to sort them. Finally, several kinds of operators such as bind join, nested loop join and hash join are used to process the results during the query evaluation step.

In this chapter, we first introduce our sampling query-based source selection which aims at both identifying relevant sources and retrieving triple patterns cardinality. Then, the query rewriting problem is outlined and a query decomposition technique that aims at pushing as much as possible of the query processing towards the remote data sources for maximal filtering of the query results is proposed. Subsequently, we propose a query sorting approach based on cardinality cost estimation and shared variables heuristics. Finally, we introduce our redundancy aware query evaluation approach.

5.2 Hybrid sources selection

This section presents our source selection strategy to both identify relevant sources for candidate triple patterns and retrieve their cardinality statistics. As discussed in section 3.3.1.3, the metadata catalogs approach is not suitable for most of the data sources. Indeed, only few SPARQL endpoints provide them and they need to be

frequently updated to be accurate. Moreover, since we are in a context of dynamic data sources related to the nature of the Web, the sources undergo frequent changes. Thus, we decide to build our own data indexing through a Sampling Query-Based approach to acquire knowledge on data sources. More specifically, instead of only sending *ASK* queries to know whether a predicate appears in a given source or not, we propose to send a *SELECT COUNT(*)* query to retrieve at the same time an estimation of the cardinality of this predicate. In this way, we build two indexes: *idxPredicateSources* and *idxPredicateCardinality*.

- The *idxPredicateSources* index records for each predicate the potential sources that are contributing to its results, in order to avoid unnecessary remote requests.

- The *idxPredicateCardinality* index provides cardinality statistics in order to estimate the triple patterns cost, which is useful during the query planning step. For each predicate the estimated cardinality will be the sum of the estimated cardinality in remote data sources ($Card(p) = \Sigma_i\ card(p, s_i)$).

The accuracy of this cardinality is a key factor of our optimization since our query planning approach is partially based on it. Thus, some less selective predicates such as *rdf:type* and *owl:sameAs* or non-selective predicates (variable as predicate for instance) need more processing. Indeed, the *SELECT COUNT* query will tend to give a high number of cardinality due to their wide use in data sources or because the predicate is a variable. Therefore, instead of sending a query with only the predicate and variables as subject and object, we replace the subject and object by their values when they are known to refine the cardinality estimation and possibly avoid source selection overestimation.

When the subject, the object and the predicate of a triple pattern are variables *(?s ?p ?o)*, we assign the Max_{Card} as cardinality to this triple pattern. Max_{Card} is a high number arbitrarily fixed to denote the maximum cardinality. For the sake of simplicity this Max_{Card} value is also assigned to predicates which are property paths operator such as the binary operator *ZeroOrMorePath (*)*. Indeed, estimating the cardinality of these kind of predicates can be very time-consuming. Therefore, a trade-off between the cardinality estimation accuracy and the processing time needs to be made.

5.2.1 Sample source selection

The following paragraph illustrates the source selection approach through an example of federated query (Q 5.1). This query retrieves the name and total population of

each departement and is evaluated over three remote data sources. The data sources S_1 and S_2 contain French geographic [28] data and the data source S_3 contains French demographic [27] data. These RDF graphs are published by the National Institute of Statistical and Economical Studies (INSEE).

Query Example 5.1: federated SPARQL query over three data sources

```
1   prefix demo: <http://rdf.insee.fr/def/demo#>
2   prefix geo:  <http://rdf.insee.fr/def/geo#>
3   select ?name ?totalPop where {
4     ?region geo:subdivisionDirecte ?dpt .
5     ?region geo:codeRegion ?v .
6     ?dpt geo:nom ?name .
7     ?dpt demo:population ?popLeg .
8     ?popLeg demo:populationTotale ?totalPop .
9   }
```

This query involves 5 triple patterns (from line 4 to line 8). The predicates (geo:subdivisionDirecte, geo:codeRegion and geo:nom) related to geographic data belong to S_1 and S_2 and the predicates (demo:population and demo:populationTotale) related to demographic data belong to S_3.

Thus, the previous source selection approach applied to Q 5.1 will generate the following indexes:

- idxPredicateSources = {
  $geo : subdivisionDirecte \rightarrow \{S_1, S_2\}$;
  $geo : codeRegion \rightarrow \{S_1, S_2\}$;
  $geo : nom \rightarrow \{S_1, S_2\}$;
  $demo : population \rightarrow \{S_3\}$;
  $demo : populationTotale \rightarrow \{S_3\}\}$

- idxPredicateCardinality = {
  $geo : subdivisionDirecte \rightarrow 3934$;
  $geo : codeRegion \rightarrow 27$;
  $geo : nom \rightarrow 41458$;
  $demo : population \rightarrow 37149$;
  $demo : populationTotale \rightarrow 37147\}$

The index *idxPredicateSources* prevents from sending the triple patterns relevant for the source S_1 and S_2 (line 4 to line 6) to the source S_3, and conversely, the triple patterns relevant for S_3 (line 6 and line 7) to S_1 and S_2.

The index *idxPredicateCardinality*, for instance, informs that the first triple pattern (line 4) is less selective than the second triple pattern (line 5) based on their cardinality. Therefore, processing the second triple pattern before and propagating afterwards the values of the variable *?region* while processing the first triple pattern will be more efficient.

5.2.2 Source selection algorithm

In this paragraph, we describe the algorithm of our source selection approach. This algorithm builds the two indexes idxPredicateSources and idxPredicateCardinality that are used to properly rewrite the initial query and to generate an optimal query plan respectively. Given a federated SPARQL query, the algorithm consists in recursively iterating over all the query expressions and for each triple pattern a SELECT COUNT query with its predicate is sent to remote data sources in order to identify the relevant ones for this predicate. In the interest of readability, the source selection algorithm is split into two algorithms. Algorithm 1 describes the recursive part whereas Algorithm 2 handles the SELECT COUNT queries.

Algorithm 1 buildIndexes iterates over the initial query expressions to build indexes. For each expression the function **initRelevantSourcesCardinality(tpSet, sourceSet)** is processed on its triple patterns

---

Input: *Query*: the initial query and *S* : the set of remote data sources

Output: idxPredicateSources and idxPredicateCardinality

foreach $(exp \in Query)$ **do**
  switch $(exp.getType())$ **do**
    `/* Union, Minus and Optional expressions */`
    case *UNION, MINUS, OPTIONAL* **do**
      $buildIndexes(exp.getLeftOp())$;
      $buildIndexes(exp.getRightOp())$;
    end
    `/* Named graph, Exists and Subquery expressions */`
    case *NAMED GRAPH, EXISTS, SUBQUERY* **do**
      $buildIndexes(exp.getExp())$;
    end
    `/* BGP with one or several triple patterns */`
    case *BGP* **do**
      $initRelevantSourcesCardinality(exp.getTriplePatterns(), S)$;
    end
  end
end

Algorithm 2 initRelevantSourcesCardinality generates SELECT COUNT queries depending on the type of predicate and builds idxPredicateCardinality (IPC) and idxPredicateSources (IPS).

Input: *triplePatternList*: A list of triple patterns,

Sources: a set of remote data sources,

nonSelectivePredicateList: a list of less selective predicates.

Output: idxPredicateSources and idxPredicateCardinality indexes

foreach $tp \in triplePatternList$ **do**
  $p \leftarrow predicate(tp)$;
  foreach $s \in Sources$ **do**
    $cardinality \leftarrow 0$;
    if $(\neg\, p.isPropertyPath())$ **then**
      if $(p.isConstant())$ **then**
        if $(nonSelectivePredicateList.contains(p))$ **then** //less selective predicate such as rdf:type or owl:sameAs
          $cardinality = getNonSelectivePredicateCardinality(s, tp)$;
        else
          $cardinality = getSelectivePredicateCardinality(s, tp)$;
        end
      else //variable as predicate such as {x ?p y}
        $cardinality = getNonSelectivePredicateCardinality(s, tp)$;
      end
    else //p is property path such as {x p1/p2 y}
      $cardinality = getPropertyPathCardinality(s, tp)$;
    end
    $updateIPCandIPS(p, s, cardinality)$;
  end
end

where:

nonSelectivePredicateList contains a list of non selective predicates which are more popular in datasets. In our case, this list contains rdf:type and owl:sameAs by default. However it can be improved by selecting the N most populated predicates for which the cardinality estimation will be refined if the values of the subject and/or the object are known.

updateIPCandIPS: associates to each predicate p its cardinality in the index IPC and adds the source s, in the set of relevant data sources the predicate p in the index IPS if $cardinality > 0$.

getSelectivePredicateCardinality: submits the SPARQL COUNT query for tp to the data source s:
SELECT (COUNT(*) AS ?number) WHERE {
SERVICE s {?subject predicate(tp) ?object } };
and return the cardinality of tp in s.

getNonSelectivePredicateCardinality: Either, submits the SPARQL COUNT query for triple pattern tp by using the value of subject(tp) and/or object(tp) when they are known. Or, returns Max_{Card} if subject, predicate and object are variables.

getPropertyPathCardinality: handles propertyPath predicates cardinality estimation and returns:

- For predicate with unary operators (zeroOrMorePath and oneOrMorePath): Max_{Card}, to avoid increasing the preprocessing time.
- unary operators (inverse and zeroOrOnePath): SPARQL COUNT query method result (getSelectivePredicateCardinality or getNonSelectivePredicateCardinality).
- binary operators alternative and sequence: the maximum or the product of cardinality respectively.

This sampling query-based approach through *SELECT COUNT* queries identifies relevant sources for each triple pattern and possibly retrieves their cardinality statistics which are crucial in the followings steps.

5.3 Hybrid query rewriting

This section is devoted to our *hybrid BGP-Triple-based* query rewriting strategy. The SPARQL federated query engines state of the art, as stated in section 3.3.2, showed

that there are two main approaches: Triple-based and BGP-based. Triple-based approach consists in evaluating the triple patterns one by one in remote data sources, retrieving the intermediate results and performing joins on the query engine side. BGP-based approach consists in grouping the triple patterns in a BGP in order to perform the intermediate results joins on the remote data servers side. The Triple-based approach is applied on horizontally partitioned data whereas the BGP-based approach is performed on vertically partitioned data.

The restriction of the BGP-based approach to vertical partitions prevents incomplete results due to missing solutions that can only be computed by joining intermediate results from different data sources. Indeed, in the case of horizontal data partitions there are two possibilities. First, results may be retrieved from a single source possibly using a BGP. Second, results may be provided by partial results from several sources which cannot be retrieved through BGPs. However, since the BGP-based strategy is more efficient than the Triple-based one, in our approach we aim at also applying it to horizontal partitions without losing results completeness. To do so, we combine both BGP-based and Triple-based approaches through heuristics that generate BGP-based sub-queries designed to maximize the parts of the query processed by the remote endpoints and Triple-based sub-queries to handle results from several sources.

The hybrid strategy described below, one of our contribution, can generate BGPs for both vertical and horizontal data partitions. The heuristics implemented aims at creating the BGPs as large as possible and applying as much as possible pattern matching at the endpoints level to minimize the number of queries generated, and minimizing the number of intermediate results collected for post processing on the federated query engine side. The larger BGPs can be evaluated independently in each data source, the more efficient the query.

A *hybrid BGP-Triple* evaluation strategy rewrites the query, computing the largest BGPs usable without alterating the results. It mixes the evaluation of BGPs and triple patterns. The BGPs generated describe local joins that can be computed in a single source. The remaining triple patterns collect a set of intermediate results over which remaining distributed join operations can be computed on the federated engine side to obtain the final results. The difficult part of hybrid strategies is to estimate the most efficient BGPs to process for each data source and the combinations of triple patterns that complete the query, from the original query and information on the distribution of data.

We use the following formalism to compute BGPs in our hybrid query evaluation strategy:

- the $G\bowtie$ operator represents all join operations (distributed and local) when evaluating a SPARQL query,

- the $L\bowtie$ operator corresponds to local joins computed by endpoints,

- the $D\bowtie$ operator corresponds to distributed joins computed by the federates query engine.

Let *TP* = $\{tp_1, \ldots, tp_n\}$ be the set of triple patterns of a BGP from the WHERE clause of a SPARQL query, *UNION* the usual set union and $\{S_1 \ldots S_m\}$ be a set of RDF data sources interfaced through SPARQL endpoints. Each result of the evaluation of *TP* is a mapping: $\{variable_i \rightarrow value_i\}$. Thus, *mappings* are combinations of the query variables to their matching values.

Each operator will retrieve mappings as follows:

- $G \bowtie_{\{S_1,S_2,\ldots,S_m\}} (TP)$ represents the join of mappings, i.e. results for TP evaluated on a *federated dataset* aggregating all data sources. It corresponds to the SPARQL query computed over the *federated dataset*, complying with the semantics described in Section 4.3:

$$G\bowtie_{\{S_1,S_2,\ldots,S_m\}} (TP) = the\ result\ evaluation\ of\ \{tp_1, \ldots, tp_n\}$$
$$in\ the\ Federated\ dataset\ (S_1, S_2, \ldots, S_m) \text{ as defined in section 4.2} \quad (1)$$

- $L\bowtie_{\{S_1,S_2,\ldots,S_m\}} (TP)$ represents the *UNION* of mappings results from the local evaluation of the BGP in each data source. This *UNION* gathers the mappings and deletes the duplicates (i.e. a SPARQL *BGP* evaluated in each source):
$L\bowtie_{\{S_1,S_2,\ldots,S_m\}} (TP) = (\{TP\}\ in\ S_1)\ )$

$$UNION\ \ (\{TP\}\ in\ S_2)$$

$$UNION\ \ \ldots\ \ (\{TP\}\ in\ S_m) \quad (2)$$

  Note that $L\bowtie\ (TP)$ is included in $G\bowtie\ (TP)$.

- $D\bowtie_{S_D} (TP)$ represents the join of mapping results from the distributed evaluation of TP (i.e. results retrieved from at least two different sources by evaluating triple patterns) with the results duplicates deletion for each triple pattern of TP:

$$D\bowtie_{S_D} (TP) = G\bowtie_{\{S_1,S_2,\ldots,S_m\}} (TP) \setminus L\bowtie_{\{S_1,S_2,\ldots,S_m\}} (TP)$$

and

$$G\bowtie_{\{S_1,S_2,...,S_m\}}(TP) = L\bowtie_{\{S_1,S_2,...,S_m\}}(TP)\ UNION\ D\bowtie_{S_D}(TP) \quad (3)$$

With $S_D = (S_{D_1}, S_{D_2}, ..., S_{D_n})$ where each S_{D_i} $(1 \leq i \leq n)$ is the set of relevant sources for tp_i $(1 \leq i \leq n)$ and $S_{D_i} \subset \{S_1, S_2, \ldots, S_m\}$.

To illustrate these operators, let us consider the execution of query Q_1 introduced in Section 2.4.3 (TP = {?team `ns:team` "SPARKS", ?team `ns:group` ?group, ?group `ns:name` ?name, ?group `ns:members` ?members}) over the two distributed data sources S_1 and S_2 described below:

Table. 5.1: Sample data sources for hybrid rewriting

S_1	S_2
{t1 ns:team "SPARKS"}	{t1 ns:team "SPARKS"}
{t1 ns:group g1}	{t1 ns:group g2}
{g1 ns:name "Modalis"}	{g2 ns:name "Wimmics"}
{g1 ns:members 12}	{g2 ns:members 9}
{t1 ns:group g3}	{g3 ns:name "MinD"}
{g3 ns:members 7}	

The 3 join operators defined above produce the following mappings when applied to the S_1 and S_2 distributed data sources in Table 5.1:

Table. 5.2: Evaluation operators mappings results

Operators	**Mappings**	**Values**	**Sources**
$L\bowtie(TP)$	μ_1	$\{?team = t_1, ?group = g_1, ?name = "Modalis", ?members = 12\}$	S_1
	μ_2	$\{?team = t_1, ?group = g_2, ?name = "Wimmics", ?members = 9\}$	S_2
$D\bowtie(TP)$	μ_3	$\{?team = t_1, ?group = g_3, ?name = "MinD", ?members = 7\}$	S_1, S_2
$G\bowtie(TP)$	μ_1	$\{?team = t_1, ?group = g_1, ?name = "Modalis", ?members = 12\}$	S_1
	μ_2	$\{?team = t_1, ?group = g_2, ?name = "Wimmics", ?members = 9\}$	S_2
	μ_3	$\{?team = t_1, ?group = g_3, ?name = "MinD", ?members = 7\}$	S_1, S_2

Table 5.2 summarizes the retrieved results for each operator:

- $L\bowtie\ (TP)$ computes a join of all 4 triple patterns of TP in each data source, producing one 4-items mapping for each of them: μ_1 from S_1 and μ_2 from S_2. However, triples associated to the group named "MinD" are distributed between the two data sources and they do not match the $L\bowtie$ local join.

- Conversely, $D\bowtie\ (TP)$ accounts for distributed mappings joins and therefore retrieves the remaining group with the mapping μ_3.

- Finally, $G\bowtie\ (TP)$ is the complete result set, resulting from the union of $L\bowtie\ (TP)$ and $D\bowtie\ (TP)$ results.

5.3.1 Global join decomposition

The $G\bowtie$ operator is associative and commutative [6, 61]:

$$\forall k < n,\ G\bowtie\ (tp_1, \ldots, tp_n) = G\bowtie\ (tp_1, \ldots, tp_k)\ .\ G\bowtie\ (tp_{k+1}, \ldots, tp_n)$$
$$= G\bowtie\ (tp_{k+1}, \ldots, tp_n)\ .\ G\bowtie\ (tp_1, \ldots, tp_k)$$

where " . " represents the binary operator for joins.

Consequently, $\forall k < n,\ G\bowtie\ (tp_1, \ldots, tp_n) =$

$$(L\bowtie\ (tp_1, \ldots, tp_k)\ UNION\ D\bowtie\ (tp_1, \ldots, tp_k))\ .$$
$$(L\bowtie\ (tp_{k+1}, \ldots, tp_n)\ UNION\ D\bowtie\ (tp_{k+1}, \ldots, tp_n)) \qquad (4)$$

This equation can be used to compute the global join operation through a combination of local and distributed joins. In the case where no source contains triples matching all triple patterns in TP, $L\bowtie\ (tp_1 \ldots tp_n)$ is empty and $G\bowtie\ (TP) = D\bowtie\ (TP)$. However, even in this case, the $G\bowtie$ operator can be computed as a join of several independent partial joins according to the distribution of triple patterns over sources. The partial joins can be computed as a union of partial $L\bowtie$ and $D\bowtie$ with subsets of $(tp_1 \ldots tp_n)$.

5.3.2 Joins generation strategy

Our hybrid evaluation strategy is based on the idea of maximizing the part of the query processed by the remote endpoints. It therefore pushes as much as possible intermediate results into the sub-queries generated to improve partial results filtering at the source. It also exploits the local and distributed join operators composition properties shown above since the partial $L\bowtie$ operators size correspond to the evaluation of the largest possible BGPs at the endpoints level. However, this

maximization is constrained by the fact that TP should be decomposed in disjoint subsets of triple patterns (rule 4 in Section 5.3.1).

The hybrid evaluation strategy also requires prior knowledge on the data sources partitioning scheme to decide on the best decomposition of triple patterns. This knowledge is provided by the source selection step through the *idxPredicateSources* index. Below, we explain the strategy of distributed and local joins generation based on the data partitioning over remote sources.

5.3.2.1 Distributed joins generation strategy

This section describes the $D\bowtie$ generation strategy. Unlike the $L\bowtie$ operators, the evaluation of $D\bowtie$ operators compute query results built with at least two distributed sources. $D\bowtie$ operators aim at handling the distributed results of horizontal partitions only, as there is no distributed join for vertical partitions.

In addition to $L\bowtie$ query results for horizontal partitions, distributed joins need to be processed in order to ensure the results completeness. For this purpose, corresponding $D\bowtie$ operators need to be generated. Thus, $L\bowtie$ generated for horizontal partitions have a corresponding $D\bowtie$ to retrieve the results provided by the distributed joins of the same triple patterns. The full result of the evaluation of triple patterns on horizontal partition is the union of the $L\bowtie$ and $D\bowtie$ as shown by the rule 3 in Section 5.3.

5.3.2.2 Local joins generation strategy

This section describes the $L\bowtie$ generation strategy given a BGP. As previously explained $L\bowtie$ operators handle joins at endpoints level.

Let *TP* a set of triple patterns as a BGP query. The $L\bowtie$ operators are generated according to the triples partitioning over data sources:

- Triple patterns of *TP* which predicates are only available in vertical partitions are used to compose a BGP (one BGP per vertical partition) that can be evaluated independently. Thus, this BGP which is also called *exclusive group* as explain in Section 3.3.2, is handled by a $L\bowtie$.

- Among the remaining triple patterns of *TP*, which are in horizontal partitions, we generate both $L\bowtie$ and $D\bowtie$. Regarding, $L\bowtie$ two criteria are considered with the aim of creating the largest possible BGPs to maximize the computation delegated to each endpoint through $L\bowtie$ operator:

  (i) The distribution of predicates inside horizontally partitioned data sources needs to be taken into account in order to generate relevant BGPs. Indeed, if two triple patterns are connected, but with their predicate not being present in the same source, there is no interest to build a *connected BGP* with these triple patterns since they will be evaluated in different sources.

  Definition 5.3.1. *A BGP is connected if every RDF term (subject or object) is connected to every RDF term of the BGP by a path of connected triple patterns. Two triple patterns are connected if they share an RDF term.*

  Two triple patterns can thus be connected through their subjects (subject-subject), their objects (object-object) and subject of the first one with the object of the second one (subject-object), and conversely, the object of the first one with the subject of the second one (object-subject).

  (ii) Two triple patterns may be grouped into a BGP only when the BGP is connected.

  A *connected BGP* enables to perform the join between triples matching the triples patterns through their shared variables, IRI or literals and therefore to reduce intermediate results. Indeed in SPARQL, joining not connected triple patterns corresponds to computing a Cartesian product between the triples matching each triple pattern. There is no interest in computing BGPs on remote servers to reduce the number of results retrieved in this case.

5.3.3 Query rewriting algorithm

In this section, we describe the main steps required to perform the hybrid query rewriting approach.

- **Step 1:** First, Algorithm 3 determines, for a given BGP, which triple patterns are horizontally or vertically partitioned.

- **Step 2:** Afterwards, for each partition a specific query rewriting strategy is performed:(i) *verticalRewrite* for vertical partitions and (ii) *horizontalRewrite*

for horizontal partitions. Besides the triple patterns, the relevant FILTER clauses are also added to generated expressions.
Definition 5.3.2. *A relevant filter is a filter which can be inserted in a BGP to be evaluated at the endpoint level.*

A filter is inserted in every BGP within which it can be evaluated following scope of filters as defined in the SPARQL semantics[1]. Indeed, according to this semantics:"*a filter is a restriction on solutions over the whole group in which the filter appears*". Thus, we insert filters in the generated BGPs in such a way that the initial query semantics is unchanged by the application of these filters. The precise criteria is that all the variables of the filter necessary for its evaluation are bound in the solution of the BGP.

The filters related to triple patterns which are not in the same BGP are processed by the federated query engine. The filters *exists* are not taken into account in *relevant filters*. Indeed, their semantics requires to consider the group graph pattern in which they occur for their evaluation. In our approach, we modify the initial group graph patterns to generate BGPs. As a consequence, it is more challenging to identify the context of filters *exists* and to check if all relevant variables are bound in the generated BGPs solutions. This issue needs more investigation. For this reason, they are also processed on the federated query engine side.

- **Step 3:** Lastly, a join expression between the two rewriting is generated to perform the query processing in such a way to produce the same results as the initial query.

Algorithm 3 focuses on BGP rewriting for the sake of readability. Similarly to Algorithm 1, this BGP query rewriting algorithm is recursively applied to each BGP for all the expressions, such as *optional, union, minus, etc.* forming the initial federated query.

The query rewriting algorithm makes use of 2 indexes:

- *idxPredicateSources* associates to each triple pattern a set of data sources hosting candidate RDF triples. It is used to determine which part of the data is horizontally and vertically partitioned. This index is an outcome of our source selection step as shown in section 5.2.1.

[1] https://www.w3.org/TR/sparql11-query/#scopeFilters

- *idxSourceTPs* associates to each remote data source, the list of triple patterns which are relevant to it (i.e. the source which contain their predicates). This index is built based on the *idxPredicateSources* index and enables to have an overview of each data source content.

  For the query Q 5.1 in section 5.2.1, we obtain the *idxSourceList* below:

  - idxSourceTPs = {
    $S_1 \rightarrow \{(?region \;\; geo:subdivisionDirecte \;\; ?dpt),$
    $(?region \;\; geo:codeRegion \;\; ?v),$
    $(?dpt \;\; geo:nom \;\; ?name)\};$
    $S_2 \rightarrow \{(?region \;\; geo:subdivisionDirecte \;\; ?dpt),$
    $(?region \;\; geo:codeRegion \;\; ?v),$
    $(?dpt \;\; geo:nom \;\; ?name)\};$
    $S_3 \rightarrow \{(?dpt \;\; demo:population \;\; ?popLeg),$
    $(?popLeg \;\; demo:populationTotale \;\; ?totalPop)\}\}$

- *idxBGPList* is based on the *idxSourceTPs* index and determines the triple patterns distribution over data sources to identify the candidate BGPs for the horizontal query rewriting. The previous index is reversed and each list of triple patterns is associated to its relevant source. In addition, sources associated to the same list of triple patterns are grouped.

  From the previous *idxSourceTPs*, the *idxBGPList* below is obtained:

  - idxBGPList = {
    $\{(?region \;\; geo:subdivisionDirecte \;\; ?dpt),$
    $(?region \;\; geo:codeRegion \;\; ?v),$
    $(?dpt \;\; geo:nom \;\; ?name)\} \rightarrow \{S_1, S_2\};$
    $\{(?dpt \;\; demo:population \;\; ?popLeg),$
    $(?popLeg \;\; demo:populationTotale \;\; ?totalPop)\} \rightarrow \{S_3\}\}$

  For the sake of clarity, we respectively note the triple patterns of the query Q 5.1 from line 4 to line 8 as $tp_1, tp_2, tp_3, tp_4,$ and tp_5. Thus, tp_1, tp_2, tp_3 are associated to S_1 and S_2 and $tp_4, \; tp_5$ are associated to S_3

From this information, the algorithm rewrites the initial federated query into a set of sub-queries preserving the query semantics. The generated sub-queries are then evaluated on the remote endpoints and their results joined by the federated query engine.

Algorithm 3 describes how BGP expressions are rewritten either as unions of $L\bowtie$ and $D\bowtie$ operators for horizontally partitioned data (Algorithm 4), or as an exclusive $L\bowtie$ for vertically partitioned data. The input for this algorithm is a $queryExp$ expression containing the list of triple patterns, *idxPredicateSources* and *idxSourceTPs* indexes and the relevant FILTERs clauses from the initial federated SPARQL query.

Algorithm 3 determines the rewriting strategy for a set of triple patterns based on the data partitioning over remote sources.

Input: queryExp, idxPredicateSources, idxSourceTPs, queryFilters

Output: the rewritten SPARQL query

$horizontalTriplePatterns \leftarrow \oslash$;

$verticalTriplePatterns \leftarrow \oslash$;

foreach $(tp \in queryExp.getTriplePatterns())$ **do**

  $p \leftarrow predicate(tp)$;

  if *(* $idxPredicateSources.get(p).size() > 1$ *)* **then**

    $horizontalTriplePatterns.add(tp)$;

  else

    $verticalTriplePatterns.add(tp)$;

  end

end

//generate the candidate BGPs index for the horizontal triple patterns

$idxBGPList \leftarrow buildBGPList(horizontalTriplePatterns, idxSourceTPs)$;

return

$evaluate(join(horizontalRewrite(idxBGPList, queryFilters),$
$verticalRewrite(verticalTriplePatterns, queryFilters)))$;

where:

verticalRewrite: rewrites triple patterns as L⋈ for each relevant source (i.e. SPARQL SERVICE clause to send this BGP to a specific source) and adds relevant filters. The triple patterns which are exclusive to the same data source are gathered in the same SERVICE clause. For instance in the sample query Q 5.1: triple patterns with predicates demo:population and demo:populationTotale are concerned.

horizontalRewrite (Algorithm 4): rewrites triple patterns as unions of L⋈ and D⋈ operators to perform the hybrid evaluation strategy. For instance in the sample query Q 5.1: triple patterns with predicates geo:subdivisionDirecte, geo:codeRegion and geo:nom are concerned.

evaluate: handles evaluation as explained in section 5.3.4.

Algorithm 4 (**horizontalRewrite**) decomposes a set of triple patterns of a BGP as a union of local and distributed expressions based on the source selection result.

Input: *ixdBGPList:* a list of BGPs annotated by their relevant sources as follows : $BGP_i \rightarrow \{s_1, ..., s_m\}$ with BGP_i a set of triple patters.

Output: *expResult* is the rewritten expression

// BGPs are sorted by the number of triple patterns

$idxBGPList.sortByDecreasingCardinality()$;

$expConnected \leftarrow \emptyset$;

$BGP\_TP\_List \leftarrow \emptyset$;

foreach *bgp in idxBGPList* **do**

  if *(*$bgp.isConnected()$*)* **then**

    // gets relevant sources for the whole BGP (local joins): $S_L \subseteq \{s_1, \ldots, s_m\}$

    $S_L \leftarrow bgp.getLocalSources()$;

    // gets relevant sources for each triple pattern of the BGP (distributed joins): $S_D = (S_{D_{tp1}}, S_{D_{tp2}}, \ldots)$

    // $S_{D_{tpi}}$: all relevant sources for tp_i (i.e. S_L + remaining relevant sources)

    $S_D \leftarrow bgp.getDistributedSources()$;

    // $L\bowtie_{S_L}$ is generated for each source in S_L (local joins)

    // $D\bowtie_{S_D}$ is generated for sources in S_D (distributed joins)

    $expUnionLandD \leftarrow union(createL\bowtie\ (S_L, bgp), createD\bowtie\ (S_D, bgp))$;

    $expConnected \leftarrow join(expConnected, expUnionLandD)$;

    // deletes in the remaining BGPs all occurrences of triple patterns handled by the current BGP

    $clean(bgp, idxBGPList)$;

    // adds tp of the current BGP to visited triple patterns list

    $BGP\_TP\_List.add(bgp)$;

  end

end

// gets not visited TPs

$free\_TP\_List \leftarrow set\_substract(queryExp, BGP\_TP\_List)$;

// generates a distributed join operator in a $D\bowtie$ expression

$S_D \leftarrow free\_TP\_List.getDistributedSources()$;

$expNotConnected \leftarrow createD\bowtie\ (S_D, free\_TP\_List)$;

// creates a join expression with visited TPs and free TPs

$expResult \leftarrow join(expConnected, expNotConnected)$;

return $expResult$;

where:

createL⋈: creates an expression evaluated through the BGP-based approach from a set of triple patterns forming a BGP and adds relevant filters from the original query for each relevant sources.

createD⋈: creates an expression to evaluate through the Triple-based approach from a set of triple patterns and adds relevant filters from the original query for a set of relevant sources.

clean: deletes the triple patterns handled by the current $L\bowtie$ and $D\bowtie$ from the list of BGPs.

By way of illustration, applied to the query Q 5.1, Algorithm 3 produces the following query expression:

$JOIN\ \{UNION\{L\bowtie_{\{S_1,S_2\}}\ (tp_1,tp_2,tp_3), D\bowtie_{(S_1,S_2)}\ (tp_1,tp_2,tp_3)\}\ ,\ \{L\bowtie_{S_3}\ (tp_4,tp_5)\}\}$ with tp_1, tp_2, tp_3, tp_4 and tp_5 respectively corresponding to the query triple patterns from line 4 to line 8 in Q 5.1.

- $L\bowtie_{\{S_1,S_2\}}$ retrieves the local results in S_1 and in S_2 for tp_1, tp_2, and tp_3.
- $D\bowtie_{\{S_1,S_2\}}$ retrieves the distributed results in S_1 and S_2 for tp_1, tp_2, and tp_3.
- $L\bowtie_{S_3}$ retrieves the local result in S_3 for tp_4 and tp_5.

This query is a simple example, in which we have one connected BGP (tp_1, tp_2, tp_3) relevant for both S_1 and S_2 sources for triple patterns horizontally partitioned without using the ***clean*** method. However in the case of more complex *idxBGPList* as proposed below, the *clean* method would have been required.

More complex query example:

Let *idxBGPList* $= \{$
$\{tp_1,tp_2,tp_3\} \rightarrow \{S_1,S_2\}, \{tp_4,tp_5\} \rightarrow \{S_3,S_4\}, \{tp_2,tp_6,tp_7\} \rightarrow \{S_5,S_6\}\}$.

We assume that all candidate BGPs are connected.

Algorithm 4 applied to this index will produce the ordered list of expressions below:

- $Exp_1 : UNION\{L\bowtie_{S_L}\ (tp_1,tp_2,tp_3), D\bowtie_{S_D}\ (tp_1,tp_2,tp_3)\}$ with $S_L = \{S_1,S_2\}$ and $S_D = (S_{D_1},S_{D_2},S_{D_3})$ where $S_{D_1} = S_{D_3} = \{S_1,S_2\}$ and $S_{D_2} = \{S_1,S_2,S_5,S_6\}$

- $Exp_2 : UNION\{L\bowtie_{S_L}(tp_4, tp_5), D\bowtie_{S_D}(tp_4, tp_5)\}$ with $S_L = \{S_3, S_4\}$ and $S_D = (S_{D_4}, S_{D_5})$ with $S_{D_4} = S_{D_5} = \{S_3, S_4\}$

- $Exp_3 : UNION\{L\bowtie_{S_L}(tp_6, tp_7), D\bowtie_{S_D}(tp_6, tp_7)\}$ with $S_L = \{S_5, S_6\}$ and $S_D = (S_{D_6}, S_{D_7})$ with $S_{D_6} = S_{D_7} = \{S_5, S_6\}$

First, in Exp_1, $S_L = S_{D_1} = S_{D_3} = \{S_1, S_2\}$ and is included in S_{D_2}. Indeed, in addition to S_L the triple pattern tp_2 is also relevant for $\{S_5, S_6\}$. Thus, these sources must be taken into account for the distributed joins operator. We can see, in Exp_3 that the triple pattern tp_2 has been removed. This is due to the fact that the triples from S_5 and S_6 matching tp_2 are already handled by $D\bowtie$ of Exp_1 with S_{D_2}. Thus, it is not necessary to keep tp_2 in Exp_3. This cleaning process is handled by the ***clean*** method in Algorithm 4. The Exp_2 is similar to the previous query.

5.3.4 Hybrid rewriting expressions evaluation

Once the hybrid query rewriting is completed, the evaluation of the generated expressions is done at two different levels. On the one hand, the $L\bowtie$ evaluation is delegated to the relevant remote data sources servers. On the other hand, the $D\bowtie$ evaluation is handled by the federated query engine itself. Algorithm 5 describes the evaluation strategy for the different expressions.

To avoid redundancy with query results already handled by $L\bowtie$, caution must be taken to ensure that the query results found during the $D\bowtie$ evaluation are not local to a source (i.e. not built with intermediate results from only one source). In order to do so, we make sure that the results generated by $D\bowtie$ come from at least two different sources. This is implemented through an algorithm (Algorithm 6) that prevents retrieving the last intermediate results of a distributed query from a given data source when all the previous ones came from this same source. This algorithm relies on an index *idxResultsSources* which associates to each partial result a set of remote sources URIs that contributed to it:

idxResultsSources: {

$R_1 \rightarrow \{S_1, S_2\}$

$R_2 \rightarrow \{S_3\}$

.....,

$R_n \rightarrow \{S_4, S_5\}\}$

We can classify the partial results of the index *idxResultsSources* in two groups:

- (i) results built from several sources: $Card(sources(R_i)) > 1$,
- (ii) results built from one source: $Card(sources(R_i)) = 1$.

When evaluating a $D\bowtie\ (tp_1, ..., tp_n)$, if the penultimate partial result is built from several sources, the last triple pattern tp_n can be evaluated in any source. However, if this result is given by one source S_i, tp_n should not be evaluated on S_i.

Furthermore, to handle the triples replication, which are side effect of the distribution and may generate unexpected additional duplicate results, we add a redundancy pruning processing in the $D\bowtie$ evaluation algorithm. To achieve this, we avoid to account replicated triple several times in *idxResultsSources* when intermediate results are retrieved from remote sources.

Algorithm 5 evaluate: resumes the evaluation strategy for the different rewritten expressions

```
result ← ∅;
foreach exp ∈ expResult do
    switch exp.getType() do
        case UNION do
            results ← join(results,
                union(evaluate(exp.leftOp()), evaluate(exp.rightOp())));
        end
        case D⋈ do
            results ← join(results,
                evaluateD⋈ (exp.getTriplePatterns());
        end
        case L⋈ do
            results ← join(results,
                evaluateL⋈ (exp.getTriplePatterns());
        end
    end
end
return result;
```

evaluateL⋈: generates a query with all triple patterns in the expression and sends it to sources containing all edges related to these triple patterns.

evaluateD⋈ (Algorithm 6): evaluates a set of triple patterns by generating a query for each triple pattern and avoids redundancy by pruning.

Algorithm 6 (**evaluateD⋈**): evaluates all triple patterns of $D\bowtie$ and avoids redundancy by pruning.

Input: idxSourceTPs, idxTPVariables, triplePatterns, sourceSet

Output: Results, the set of SPARQL query results.

$Result \leftarrow \emptyset$;
foreach $tp \in triplePatterns$ **do**
  $newResults \leftarrow \emptyset$;
  foreach $s \in tp.getSources()$ **do**
    if $(\neg\, tp.isLastPattern())$ **then**
      $newResults+ = process(Results, tp, s)$;
    else // last pattern: handle the pruning
      $partialResultToPrune \leftarrow \emptyset$;
      foreach $R_i \in Results$ **do**
        // partial results to prune for s because already handled by the corresponding $L\bowtie$
        if $(R_i.sources.card() = 1)$ & $(R_i.sources.contains(s))$ **then**
          $partialResultToPrune.add(R_i)$;
        end
      end
      $copiedResults \leftarrow Results.copy()$;
      $relevantPartialResult \leftarrow copiedResults.remove(partialResultToPrune)$;
      $newResults+ = process(relevantPartialResult, tp, s)$;
    end
  end
  $Results \leftarrow newResults$;
end
return $Results$;

where:

process (Results, tp, s):
$tmp \leftarrow execQuery(Results, tp, s)$;
$tmp' \leftarrow bookKeeping.add(tmp, s)$; //skips replicated triples and saves the source
$newResults \leftarrow join(Results, tmp')$;
return $newResults$;

bookKeeping: handles the history of results to determine if all results came from the same source in order to avoid redundancy. Also avoids accounting for replicated triples several times in results.

In general, the query rewriting algorithm produces a set of sub-queries with varied processing time as previously shown in section 3.3.3. Therefore, before the evaluation, sub-queries planning is necessary to efficiently evaluate federated queries.

5.4 Sub-queries sorting optimization

In this section we describe the cost-based query sorting we use to optimize the sub-queries planning. Instead of using only heuristics on the federated query patterns through their number of variables or their shared variables or arbitrary giving priority to some query patterns such as SERVICE clauses, we are seeking to estimate the query expressions cost based on the cardinality statistics we retrieved during the source selection step. We first estimate the cost for all the query expressions and build the *idxExpCost* index. Algorithm 7 creates this index. Based on it, Algorithm 8 sorts the whole query. Besides the cost estimation, we also take into account the links between expressions. Indeed since we are propagating intermediate results to the following expressions, when two expressions share several variables, IRIs, or literals, evaluating these two expressions consecutively may be more efficient than evaluating another with lower estimated cost. This heuristics is similar to *Free Variables Counting (FVC)* and *Shared Variables (SV)* described in section 3.3.3. Finally, when two expressions have the same cost and are not linked to the previous one we give the priority to the one with filters, when appropriate. Therefore, we combine both cost estimation and heuristics on query patterns.

5.4.1 Cost estimation algorithm

The aim of the cost estimation algorithm is to statically estimate the initial expressions cost based on the cardinality statistics retrieved for each triple pattern during the source selection step. We also rely on heuristics based on the simplified semantics below which does not take into account the whole results cardinality as defined in the SPARQL semantics[2]. We use the compatibility between solution mappings terminology, written $\mu_1 \sim \mu_2$. The incompatibility between solution mappings terminology is denoted by $\mu_1 \nsim \mu_2$. Let Ω_1 and Ω_2 be sets of mappings; the join, union, minus, and optional operations for Ω_1 and Ω_2 are defined as follows [14]:

- $\Omega_1 \; Join \; \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \; \mu_2 \in \Omega_2 \; and \; \mu_1 \sim \mu_2\}$,
- $\Omega_1 \; Union \; \Omega_2 = \{\mu \mid \mu \in \Omega_1 \; or \; \mu \in \Omega_2\}$,
- $\Omega_1 \; Minus \; \Omega_2 = \{\mu \in \Omega_1 \mid \forall \, \mu_0 \in \Omega_2 : \mu \nsim \mu_0\}$,
- $\Omega_1 \; Optional \; \Omega_2 = (\Omega_1 \; JOIN \; \Omega_2) \cup (\Omega_1 \; MINUS \; \Omega_2)$

[2] https://www.w3.org/TR/sparql11-query/#sparqlAlgebra

In our approach, based on the previous definitions we estimate the initial expressions cost as follows:

- Triple pattern : we use the estimated cardinality in the index *idxPredicateCardinality* as cost for triple pattern.

- BGP: the BGP expression results cardinality is obtained by the results cardinality of each triple pattern. However, this is the cardinality in case of Cartesian product between triple patterns mappings. Since we generally generate connected BGPs and are propagating the known values for the following triple patterns, we decide to use the minimum estimated cost of the triple patterns as BGP estimated cost.

- Union: the cost of a union of two expressions is the sum of the cost of the two expressions.

- Minus: the minus expression cardinality will not exceed its first expression cardinality therefore we use the estimated cost of the first expression as its cost expression.

- Optional: the result of the optional expression is the union of the mappings of:

  - the join between the two expressions mappings and,

  - the difference between these two mappings.

    Based on the previous heuristics each part cost is estimated by the first expression cost. therefore, we also estimate the optional cost as the estimated cost for its first expression.

Algorithm 7 buildExpCost iterates over the query expressions and recursively estimates their cost based on the index *idxPredicateCardinality*

Input: *Query*: the initial query and S : the set of remote data sources
idxPredicateCardinality: the predicate cardinality index built during sources selection.
Output: idxExpCost

```
foreach (exp ∈ Query) do
    expCost ← 0;
    switch (exp.getType()) do
        case UNION do
            expCost = idxExpCost.get(exp.getLeftOp())
             + idxExpCost.get(exp.getRightOp());
        end
        case OPTIONAL, MINUS do
            expCost = idxExpCost.get(exp.getLeftOp());
        end
        case BGP do
            expCost = exp.getMinCostTP();
        end
        case NamedGraph do
            expCost = idxExpCost.get(exp.getFirstExp());
        end
    end
    idxExpCost.put(exp, expCost);
end
return idxExpCost;
```

5.4.2 Cost-based and shared links sorting approach

This algorithm combines the estimated cost for the query expressions and heuristics on query patterns through the number of links between expressions and the number of connected expressions.

Algorithm 8, recursively determines the expression with the lowest cost from the query expressions list and appends it to the sorted query expressions list. A selected expression, which is not a triple pattern, is also sorted. Once an expression is appended, its linked expressions are also appended to the sorted query expressions list.

Algorithm 8 sortQuery: performs the sub-queries sorting based on their estimated cost.

Input: idxExpCost, queryExpList (the query expressions list)

Output: *querySortedExpList*: the sorted list of the query expressions

```
querySortedExpList ← ∅;
while (¬ queryExpList.isEmpty()) do
    currentExp ← getMinExpCost(idxExpCost, queryExpList);
    if (¬ querySortedExpList.contains(currentExp)) then
        if (¬ currentExp.isTriple()) then
            tmpSortedExpList ← sortQuery(idxExpCost, currentExp.getExpList());
            currentExp.setExpList(tmpSortedExpList)
        end
        querySortedExpList.add(currentExp);
        queryExpList.remove(currentExp);
        connectExpList ← getConnectedExp(currentExp, queryExpList);
        foreach (relatedExp ∈ connectExpList) do
            if (¬ querySortedExpList.contains(relatedExp)) then
                sortQuery(idxExpCost, relatedExp.getExpList());
                querySortedExpList.add(relatedExp);
                queryExpList.remove(relatedExp);
            end
        end
    end
end
return querySortedExpList;
```

where:

getMinExpCost: gives the expression with the lowest cost through (idxExpCost). When two expressions have the same cost, the one with filter (or more filters) will be selected.

getConnectedExp: gives the list of expressions linked to the current expression from the current query expressions list. This list is sorted by the number of shared links (variables, IRIs or literals) and the number of linked expressions. Indeed, we want to evaluate a succession of expressions sharing links because we propagate the intermediates results into the following expressions. *getconnectedExp* returns a list of expressions sorted in such a way that the first expression is the one sharing more links with the current expression and then related to more expressions.

5.5 Query evaluation

At this step, the federated query is evaluated following the previous query plan. First, sub-queries are progressively sent to the remote sources. Then, intermediate results are retrieved by the query engine and remaining joins are performed to build the final query results. As shown in section 3.3.4, different kinds of joins operators such as nested loop join or bind join can be used. In this section, we describe our query evaluation based on two strategies: (i) using variable *bindings* for joins processing as much as possible, and (ii) exploiting parallelism to enhance the remote requests processing.

5.5.1 Bindings strategy

This strategy aims at exploiting the values of variables already known from the intermediate results of previous query expressions in the following query expressions with the same variables. Each time we have a partial results and we are sending query expressions to the remote servers, we add to these expressions the relevant bindings from the partial results. The processed expressions can either be a single triple pattern, or a BGP depending on the evaluation strategy. Both *VALUES* or *FILTER* SPARQL clauses can be used to constrain the remote evaluation of subsequent expressions with bindings. Therefore, *FILTER* and *VALUES* expressions in the initial federated query can also be considered as *bindings*. The binding strategy produces more selective triple patterns or BGPs and therefore reduces the partial results transferred from the remote data sources.

5.5.2 Parallelism strategy

This strategy consists in using parallelism when querying the remote data sources to enable more asynchronous distributed evaluation of federated queries. Let L the list of sub-queries resulting from the query rewriting processing step. First, the different sub-queries are sequentially evaluated to take advantage of the bindings strategy as explained above. Furthermore, the expressions of this sequence are to be executed on several remote endpoints and the execution of an expression on these endpoints is performed in parallel. The federated engine sends the expression to the remote endpoints in parallel. Then the aim is to concurrently query the endpoints of data sources for each sub-query of L which can be a single triple pattern or a BGP depending on the evaluation strategy ($D\bowtie$ or $L\bowtie$). The federated query engine handles the partial results as soon as they are retrieved from the endpoints. Algorithm 9 below illustrates the parallelism evaluation of remote endpoints.

Algorithm 9 Parallel federated evaluation

Input: *endpointList*: the list of endpoints,
scheduler: a thread pool handling the parallel execution,
queryExpList: the list of query expressions,
bindingsList: the list of bindings.
Output: *Results*: the set of SPARQL query results

```
foreach (exp ∈ queryExpList) do
    // sends remote requests to relevant sources for exp in parallel with its relevant
     bindings
    foreach (e ∈ exp.endpointList()) do
        scheduler.submit(e.execute(exp, exp.getBindings(bindingsList)));
    end
    // retrieves partial results when task are completed
    foreach task ∈ scheduler.getFinished() do
        Results+ = cleanDuplicate(task.getResult());
    end
end
```

5.6 Conclusion

This chapter aimed at efficiently addressing SPARQL federated query processing while transparently querying distributed data sources and taking into account the data replication and results completeness challenges. To achieve this, we proposed different strategies to enhance each step of the federated query processing approach, namely source selection, query rewriting, query planing and query evaluation.

First, we proposed a Sampling Query-Based approach for the source selection step to acquire knowledge on data sources. The Sampling Query-based approach proposed relies on *SELECT COUNT* queries which enable to both identify relevant sources for the initial query triple patterns and to retrieve an estimation of the cardinality of the predicates when possible. At this step we build two indexes: *idxPredicateSources* and *indxPredicateCardinality*. The former keeps the relevant sources for each predicate and gives information on data distribution, and the latter is useful for the query planing step.

Afterwards, we introduced a hybrid BGP-Triple query rewriting approach. The aim of this approach is to maximize the part of the query processing handled by the endpoints in order to reduce network cost communication (remote requests and transferred data) while maintaining the results completeness. In that perspective, we introduced some heuristics to generate efficient BGPs for both horizontal and

vertical data partitions to handle local joins and to restrict Triple-based evaluation to only distributed joins.

Then, based on statistics on predicates retrieved during the sources selection step, we proposed to improve the query sorting by combining the cost estimation and heuristics on query patterns sorting during the query rewriting process. In addition to the estimated cost of each query pattern, the links between query expressions are also considered in our sorting strategy. The choice of the last criterion is explained by the fact that we are using bindings during the query evaluation phase. Indeed, in the query evaluation step, we used bindings to propagate already known values in order to reduce intermediate results from remote data sources. Besides, we also used parallelism to concurrently query remote endpoints. Finally, our evaluation strategy avoids to generate duplicate results which are side effect of triple replication.

In summary, we proposed both static and dynamic strategies to improve the federated query processing efficiency. The static optimizations are performed during query rewriting and planning steps whereas the dynamic optimizations are achieved during the query evaluation. In the following chapter we will demonstrate the impact of these different strategies on federated query processing performance.

6 Implementation and Evaluation

Contents

6.1 Introduction **80**
6.2 Implementation **80**
6.3 Evaluation **82**
6.3.1 Hybrid Data Partitioning (HDP) experiment 82
6.3.2 FedBench benchmarck 89
6.3.3 KGRAM-DQP vs FedX 96
6.4 Conclusions **100**

6.1 Introduction

We previously introduced several optimization strategies for SPARQL main federated query steps. In this chapter we first briefly present the KGRAM-DQP engine that supports our federated query processing approach and optimization strategies. Afterwards, we propose several experiments to assess the impact of these strategies with regard to the challenges related to federated query processing such as results completeness, data replication, and query processing efficiency. The first experiment compares KGRAM-DQP, instrumented with the different optimizations proposed, with its initial version. The second experiment compares the optimized KGRAM-DQP with competitor SPARQL federated query engines from the state-of-the-art.

6.2 Implementation

The implementation of our federated query processing approach is performed on the Knowledge Graph Abstract Machine (KGRAM) framework. KGRAM [20, 21] is part of the Corese [22, 19] Semantic Web framework which enables users to represent RDF data, to query this data through SPARQL queries and to reason on it. KGRAM can be used for both data providers and data queriers. On the one hand, providers can expose their data through a SPARQL endpoint and interpret SPARQL queries. On the other hand, queriers can use KGRAM to query any remote data source endpoint through a *Producer*. A *Producer* is the query engine interface for data sources that sends sub-queries to remote data source endpoints. More generally, *Producers* handle data sources heterogeneity. Indeed, KGRAM introduced *Abstract Knowledge Graphs (AKG)* which allow on the one hand to represent and to query semantic data and on the other hand to address heterogeneous data through graph-based view representation of targeted databases. A query received by the KGRAM engine is transformed into an *abstract query language (AQL)*, independent of specific data source types, to be evaluated on Abstract Knowledge Graphs. The Producers use AKGs and AQLs as pivot representations and act as mediators for heterogeneous data sources.

KGRAM-DQP, is an extension of KGRAM that handles federated querying over several distributed data sources through a *MetaProducer*, an interface to several producers that implements parallel querying. Two main parallelism approaches are implemented by the *MetaProducer*:

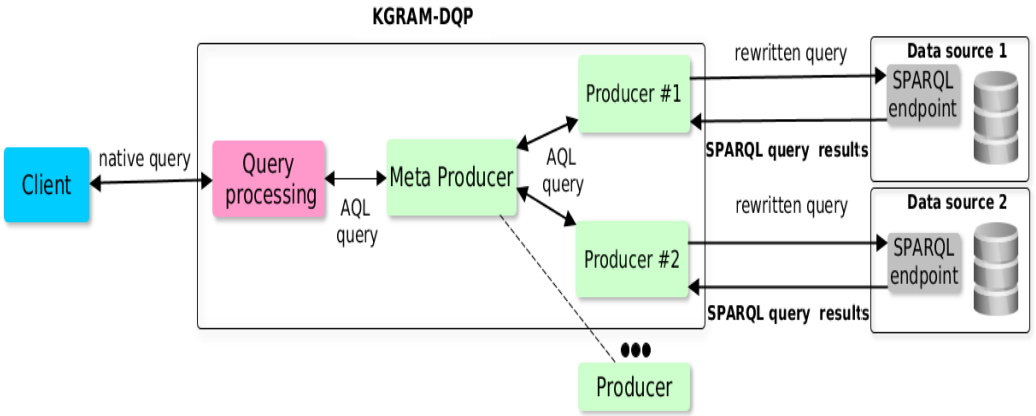
- Parallel-wait approach : this strategy consists in querying the data sources through Producers in parallel and waiting for the complete query results retrieval of all Producers through a synchronization barrier for each sub-query.

- Parallel-pipeline approach: this approach queries the data sources in parallel but processes the query results as soon as they are produced by the Producers instead of waiting all Producers to finish their processing.

The Parallel-pipeline approach is more efficient than the Parallel-wait approach, since query results are processed as soon as they are available. However, its impact is restricted because the KGRAM evaluation engine on which relies KGRAM-DQP does not make use of a full asynchronous query processing approach that would be needed to take the full advantage of this approach.

KGRAM has a local *Producer* interface for each remote endpoint. All producers are interfaced to the core engine through a *MetaProducer* as illustrated in Figure 6.1 that depicts the architecture of the KGRAM-DQP federated query engine. The query processing component handles the different steps of the federated query processing: source selection, query rewriting, query planing and query evaluation.

Figure. 6.1: KGRAM-DQP architecture



6.3 Evaluation

6.3.1 Hybrid Data Partitioning (HDP) experiment

6.3.1.1 Material

The two experimental data sets used are French geographic[1] and demographic[2] RDF graphs published by the National Institute of Statistical and Economical Studies (INSEE). Both contain linked data on the geographical repartition of population on the French administrative territory decomposition. These data are not natively available through SPARQL endpoints, therefore we used KGRAM to expose these RDF graphs.

To reproduce a context of both vertical and horizontal partitioning of data, the demographic data set is in a vertical partition (source S_1) and the geographic data set is horizontally partitioned in three test cases:

- P_1 **partitioning** (data duplication): the geographic data is duplicated in two distributed sources (S_2 and S_3), each containing a whole copy of the data. This test case aims at evaluating the handling of redundant results by each query evaluation method.

- P_2 **partitioning** (global distribution of predicates): the geographic data is partitioned into two sources (S_4 and S_5), each containing all predicates but not all triples related to these data.

- P_3 **partitioning** (partial distribution of predicates): the geographic data is partitioned in three sources (S_6, S_7 and S_8), each containing a subset of predicates related to these data.

The first partitioning aims at testing the handling of redundant results by all query evaluation methods, and the two last ones aim at testing the global and partial distribution strategies introduced in Section 5.3.2.2. The data partitions are summarized in Table 6.1.

[1] INSEE geographic data set: http://rdf.insee.fr/geo/2014/cog-2014.ttl.zip
[2] INSEE demographic data set: http://rdf.insee.fr/demo/popleg-2013-sc.ttl.zip

Table. 6.1: Data sets partitioning

Datasets	Predicates	File size (Turtle)	Nb Triples
S_1: Demographic dataset	population , populationTotale	17,9 Mo	222 429
S_2: Geographic dataset	codeRegion, subdivisionDirecte, nom	19,9 Mo	368 761
S_3: Geographic dataset copy	codeRegion, subdivisionDirecte, nom	19,9 Mo	368 761
S_4: Geographic dataset part 1	codeRegion, subdivisionDirecte, nom	19,2 Mo	351 720
S_5: Geographic dataset part 2	codeRegion, subdivisionDirecte, nom	749,4 ko	17 217
S_6: Geographic dataset part 2.1	codeRegion, subdivisionDirecte	735,3 ko	16 963
S_7: Geographic dataset part 2.2	codeRegion, nom	15,5 ko	232
S_8: Geographic dataset part 2.3	subdivisionDirecte, nom	33,9 ko	567

6.3.1.2 Methods

Six queries, shown below, where selected as a representative set of SPARQL queries covering the most common clauses:

1. Q_{SELECT}: is made of a simple SELECT clause
2. Q_{UNION}: introduces a UNION clause
3. Q_{MINUS}: introduces a query with MINUS negation
4. Q_{FILTER}: introduces a query with several filters
5. Q_{OPT}: introduces a query with an OPTIONAL clause
6. Q_{ALL}: is a combination of all foregoing clauses

In the queries below, the geographical predicates and the demographical predicates are prefixed as follows:

```
PREFIX  geo:  <http://rdf.insee.fr/def/geo#>
PREFIX  demo: <http://rdf.insee.fr/def/demo#>
```

The queries are variations around listing the population count in diverse sub-geographical areas.

Queries:

Query Q_{SELECT}

```
SELECT ?name ?totalPop WHERE {
 ?region geo:codeRegion ?v .
 ?region geo:subdivisionDirecte ?dpt .
 ?dpt geo:nom ?name .
 ?dpt demo:population ?popLeg .
 ?popLeg demo:populationTotale ?totalPop .
} ORDER BY ?totalPop
```

Query Q_{UNION}

```
SELECT ?district ?totalPop WHERE {
 { ?region geo:codeRegion ?v .
   ?region geo:subdivisionDirecte ?dpt .
   ?dpt geo:nom ?name .
   ?dpt geo:subdivisionDirecte ?district .
   FILTER (?v <= "42") }
 UNION {
   ?region geo:codeRegion ?v .
   ?region geo:subdivisionDirecte ?dpt .
   ?dpt geo:nom ?name .
   ?dpt geo:subdivisionDirecte ?district .
   FILTER (?v > "42") }
 ?district demo:population ?popLeg .
 ?popLeg demo:populationTotale ?totalPop .
} ORDER BY ?totalPop
```

Query Q_{MINUS}

```
SELECT ?name ?totalPop WHERE {
 ?region geo:codeRegion ?v .
 ?region geo:subdivisionDirecte ?dpt .
 ?dpt geo:nom ?name .
 ?dpt geo:subdivisionDirecte ?district .
 MINUS {
  ?region geo:codeRegion "24" .
  ?dpt geo:subdivisionDirecte
    <http://id.insee.fr/geo/arrondissement/751>
 }
 ?district demo:population ?popLeg .
 ?popLeg demo:populationTotale ?totalPop .
} ORDER BY ?totalPop
```

Query Q_{FILTER}

```
SELECT ?district ?cantonNom WHERE {
 ?region geo:codeRegion ?v .
 ?region geo:subdivisionDirecte ?dpt .
 ?dpt geo:nom ?name .
 ?dpt geo:subdivisionDirecte ?district .
 ?district geo:subdivisionDirecte ?canton .
 ?canton geo:nom ?cantonNom .
 FILTER (?v = "11")
 FILTER (?cantonNom = "Paris␣14e␣␣canton")
 ?dpt demo:population ?popLeg .
 ?popLeg demo:populationTotale ?totalPop .
} ORDER BY ?totalPop
```

Query Q_{OPT}

```
SELECT * WHERE {
 ?region geo:codeRegion ?v .
 ?region geo:subdivisionDirecte ?dpt .
 ?dpt geo:nom ?name .
 OPTIONAL { ?dpt geo:subdivisionDirecte ?district }
 ?dpt demo:population ?popLeg .
 ?popLeg demo:populationTotale ?totalPop .
}
```

Query Q_{ALL}

```
SELECT ?name ?totalPop WHERE {
 { ?region geo:codeRegion  "24".
   ?region geo:subdivisionDirecte ?dpt .
   ?dpt geo:subdivisionDirecte ?district .
   OPTIONAL { ?district geo:nom ?name }
 } UNION {
   ?region geo:codeRegion ?v .
   ?region geo:subdivisionDirecte ?dpt .
   ?dpt geo:subdivisionDirecte ?district .
   ?district geo:nom ?name .
   MINUS {
    ?region geo:codeRegion ?v .
    ?dpt geo:subdivisionDirecte <http://id.insee.fr/geo/arrondissement/751> .
    ?district geo:subdivisionDirecte  <http://id.insee.fr/geo/canton/6448> .
    FILTER (?v = "24")
   }
 }
 ?district demo:population ?popLeg .
 ?popLeg demo:populationTotale ?totalPop .
} ORDER BY ?totalPop
```

Several metrics are reported for each experiment to measure the query computing efficiency, both in term of query processing time and in number of remote requests to endpoints:

- The Query processing time: the total query processing time as measured on the federated query engine side (from source selection to query results delivery). This metric assesses the querying performance.

- The number of remote requests: measured as the number of requests sent to endpoints for evaluation. This metric evaluates the amount of messages exchanged and the resulting network load.

In each run, our hybrid strategy (optimized KGRAM-DQP), that implements BGP-based evaluation on both vertical and horizontal data partitions, is compared to the reference implementation (basic KGRAM-DQP), that implements a triple-based evaluation strategy with BGP generation for vertical partitions only, similarly to other state-of-the-art DQP engines. The correctness of the hybrid algorithm is first tested by verifying that all hybrid requests produce exactly the same results as their reference counter-part. The evaluation of our optimized KGRAM-DQP implementation addresses both the completeness of the results (with a minimal number of sub-queries processing), and query execution performance. The experiments proposed in this section aim at demonstrating the completeness of the results independently from the data partitioning scheme and the kind of SPARQL ”select” query executed. They are based on the querying of two linked RDF datasets, and a set of representative SPARQL queries covering most common SPARQL clauses. In each case, performance is measured in terms of number of remote requests generated and computation time. All experiments are run on a single dedicated quad-core laptop (Dell Latitude E6430 running Linux Ubuntu 14.04, 2.7 GHz Intel CPU i7-3740QM, 8 GB RAM) running all SPARQL endpoints and the KGRAM query engine, thus preventing any impact from the network load on performance measurements. To further alleviate any problem related to execution time variations that cannot be controlled in a multi-core multi-threaded execution environment, each experiment is executed 6 times and computation times are averaged.

6.3.1.3 Performance and results

Figure 6.2 displays the performance (processing time) and workload (number of remote requests performed) for each test query and each partitioning scheme. The execution times shown in Figure 6.2 (top) are averaged executions of 6 runs and error bars represent $\pm$ 1 standard deviation.

Figure 6.2 (top) displays groups of measurements for each query, shown in the following order: SELECT, UNION, MINUS, FILTER, OPTIONAL, ALL. Each group of 6 measurements shows the Basic and the Optimized implementations of KGRAM-DQP processing time for partitioning schemes P_1 (duplication), P_2 (global distribution) and P_3 (partial distribution). Similarly, Figure 6.2 (bottom) shows the number of remote requests processed for each test query. Measurements are further analyzed by partitioning scheme.

As can be seen, the Optimized strategy is consistently faster than the reference strategy in all cases, with execution time reduced by 1.9% to 76% depending on the case. Similarly, the number of remote requests processed is consistently reduced.

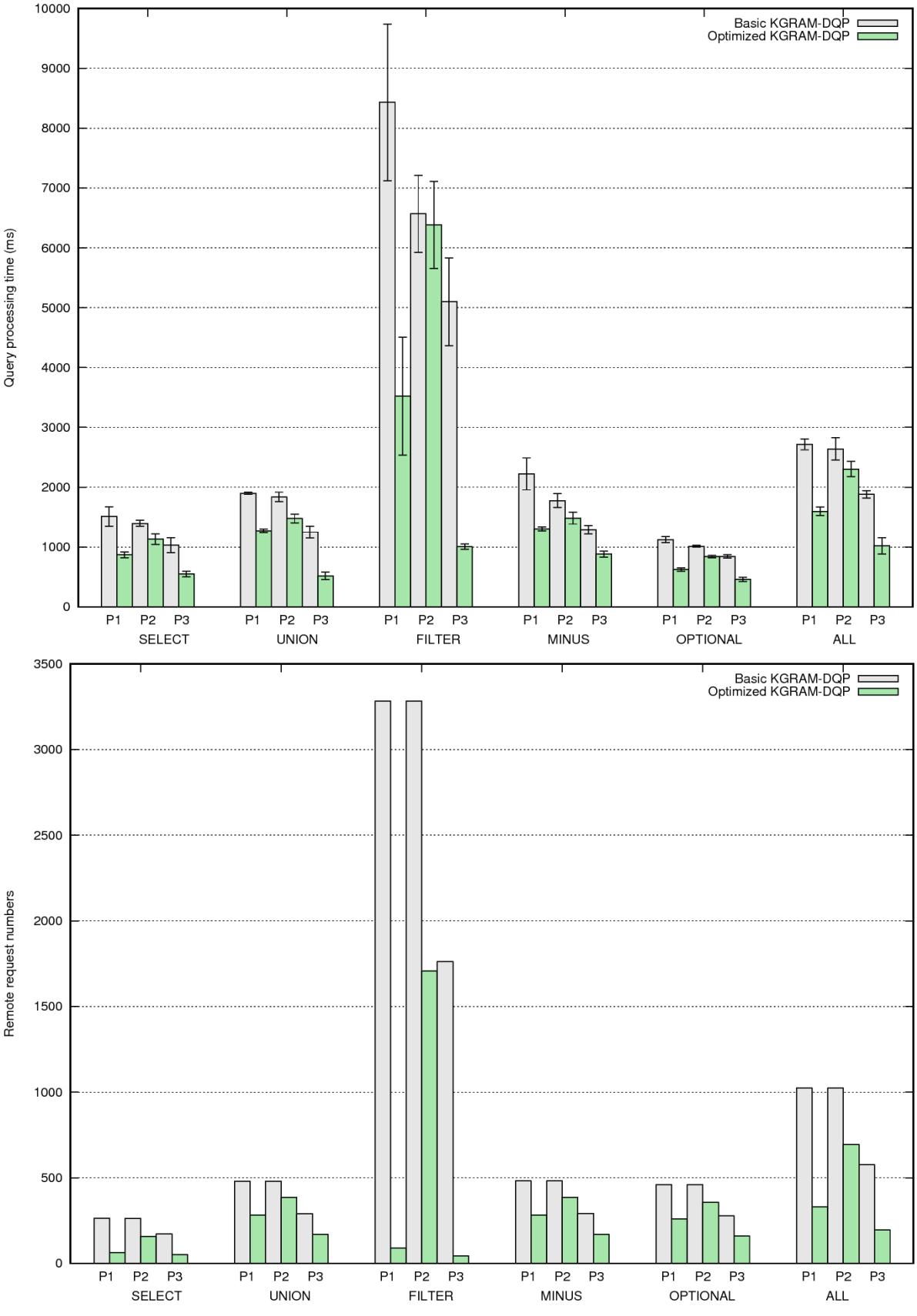
The impact of the hybrid approach depends on the efficiency of BGP-based evaluation which, in turn, depends on the distribution of the data queried. Thus, the more $L\bowtie$ retrieved results compared to $D\bowtie$, the more efficient the Optimized KGRAM-DQP approach.

In the P_1 partitioning scheme, all data are duplicated in two sources, which means the local joins ($L\bowtie$) will retrieve all the final results. However, the distributed joins ($D\bowtie$) processing is initiated to handle the possible distributed triples and finally deleted by the pruning algorithm to tackle the distributed redundancy issue. Even with this additional processing time, the Optimized implementation reduces the processing time by 33% to 53% and the number of remote requests by 41% to 97% compared to the Basic implementation depending from the fact $D\bowtie$ operator avoids to reproduce results already given by $L\bowtie$.

In the P_2 partitioning scheme, there is no data duplication, and data predicates are globally distributed. More results are retrieved through distributed joins processing than local joins processing. Execution times for the Optimized engine are consequently higher than in the previous case. The experiments still show a reduction of the processing time by 2.1% to 20% and a reduction of the number of remote requests by 19% to 48%.

Owing to partial distribution, the P_3 partitioning scheme exhibits smaller expressions (partial BGPs) than in the P_2 case (global BGP) in term of number of triple patterns, with a higher chance for matching more intermediate results through $L\bowtie$. This explains why the results of the Optimized approach with P_3 is more efficient than the results with P_2. Furthermore, since the distributed joins are less selective in this case, the pruning algorithm is applied earlier than in the P_1 case, which explains why results are improved. In this case, the hybrid approach reduces the processing time by 31% to 80% and the number of sub-queries by 41% to 97% between Optimized KGRAM-DQP and Basic KGRAM-DQP.

Figure. 6.2: Top: query processing time. Bottom: remote requests sent



6.3.2 FedBench benchmarck

This experiment aims at comparing the Optimized KGRAM-DQP with FedX, the most competitive federated query processing engine [30, 71], through the FedBench benchmark. The FedBench [73] benchmark is dedicated to federated query engines processing and performance.

6.3.2.1 Material and methods

FedBench proposes several datasets[3] and a set of federated queries[4] over these data sources. In this experiment we focus on "Life Science" queries (LS1 to LS7) evaluated over 4 remote data sources (in Table 6.2):

- DBPedia subset: a subset of DBPedia
- ChEBI: Chemical Entities of Biological Interest
- DrugBank: DrugBank bioinformatics and cheminformatics dataset describing drugs and drug targets through a pharmacological perspective
- KEGG: Kyoto Encyclopedia of Genes and Genomes

Datasets are related through the *keggCompoundId* predicate (between DrugBank and KEGG) and the owl:sameAs predicate. The datasets are exposed through Virtuoso SPARQL endpoints[5]. Similarly to the former HDP experiment (Section 6.3.1.1), the virtuoso [25] SPARQL endpoints and the KGRAM-DQP engine are executed on the same Linux machine (Dell Latitude E6430 running Linux Ubuntu 14.04, 2.7 GHz Intel CPU i7-3740QM, 8 GB RAM).

Table. 6.2: FedBench Datasets

Datasets	Version	Domain	Number of Triples
DBPedia subset	3.5.1	Generic	43.6M
KEGG	2010-11-25	Chemicals	1.09M
Drugbank	2010-11-25	Drugs	767k
ChEBI	2010-11-25	Compounds	7.33M

[3] https://code.google.com/archive/p/fbench/wikis/Datasets.wiki
[4] https://code.google.com/archive/p/fbench/wikis/Queries.wiki
[5] https://github.com/openlink/virtuoso-opensource

Queries: for the sake of clarity, the following namespace prefixes are used:

```
PREFIX  rdf <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  owl: <http://www.w3.org/2002/07/owl#>
PREFIX  drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX  drugbank-d: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>
PREFIX  drugbank-c: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugcategory/>
PREFIX  dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX  dbpedia-owl-drug: <http://dbpedia.org/ontology/drug/>
PREFIX  kegg: <http://bio2rdf.org/ns/kegg#>
PREFIX  chebi: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX  purl: <http://purl.org/dc/elements/1.1/>
PREFIX  bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
```

The 7 life science queries of FedBench are depicted below:

Query Q_{LS1}

```
SELECT ?drug ?melt WHERE {
{ ?drug drugbank:meltingPoint ?melt . }
UNION
{ ?drug dbpedia-owl-drug:meltingPoint ?melt . }
}
```

Query Q_{LS2}

```
SELECT ?predicate ?object WHERE {
{ drugbank-d:DB00201 ?predicate ?object . }
UNION
{ drugbank-d:DB00201 owl:sameAs ?caff .
  ?caff ?predicate ?object . }
}
```

Query Q_{LS3}

```
SELECT ?Drug ?IntDrug ?IntEffect WHERE {
?Drug rdf:type dbpedia-owl:Drug .
?y owl:sameAs ?Drug .
?Int drugbank:interactionDrug1 ?y .
?Int drugbank:interactionDrug2 ?IntDrug .
?Int drugbank:text ?IntEffect .
}
```

Query Q_{LS4}

```
SELECT ?drugDesc ?cpd ?equation WHERE {
?drug drugbank:drugCategory drugbank-c:cathartics .
?drug drugbank:keggCompoundId ?cpd .
?drug drugbank:description ?drugDesc .
?enzyme kegg:xSubstrate ?cpd .
?enzyme rdf:type kegg:Enzyme .
?reaction kegg:xEnzyme ?enzyme .
?reaction kegg:equation ?equation .
}
```

Query Q_{LS5}

```
SELECT ?drug ?keggUrl ?chebiImage WHERE {
?drug rdf:type drugbank:drugs .
?drug drugbank:keggCompoundId ?keggDrug .
?keggDrug bio2rdf:url ?keggUrl .
?drug drugbank:genericName ?drugBankName .
?chebiDrug purl:title ?drugBankName .
?chebiDrug chebi:image ?chebiImage .
}
```

Query Q_{LS6}

```
SELECT ?drug ?title WHERE {
?drug drugbank:drugCategory drugbank-c:micronutrient .
?drug drugbank:casRegistryNumber ?id .
?keggDrug rdf:type kegg:Drug .
?keggDrug bio2rdf:xRef ?id .
?keggDrug purl:title ?title .
}
```

Query Q_{LS7}

```
SELECT ?drug ?transform ?mass WHERE {
?drug drugbank:affectedOrganism 'Humans and other mammals' .
?drug drugbank:casRegistryNumber ?cas .
?keggDrug bio2rdf:xRef ?cas .
?keggDrug bio2rdf:mass ?mass
OPTIONAL { ?drug drugbank:biotransformation ?transform . }
FILTER ( ?mass > 5 )
}
```

6.3.2.2 Query rewriting and query sorting optimizations impact

In this section, we illustrate the impact of the query rewriting and sorting optimizations based on the source selection step, which identifies relevant data sources for each triple pattern.

The optimized queries of the 7 life science queries by KGRAM-DQP are depicted below:

tp_i corresponds to the triple patterns from the *WHERE* clause of the queries.

- $Q_{LS1_{optimized}} = L \bowtie_{DrugBank} (tp_1)$

  $Q_{LS1_{optimized}}$ highlights the importance of the source selection. Since only one relevant data source is associated to the triple pattern of the first argument of the *UNION*, Q_{LS1} is reduced to one triple pattern sent to the data source *DrugBank*.

- $Q_{LS2_{optimized}} = UNION\{L \bowtie_{DrugBank} (tp_1), JOIN\{L \bowtie_{DrugBank} (tp_2), D \bowtie_{S_D} (tp_3)\}\}$
  with $S_D = (S_{D_{tp3}})$ *and* $S_{D_{tp3}} = \{DrugBank, DBPedia\ subset, KEGG, ChEBI\}$

  $Q_{LS2_{optimized}}$ stresses the necessity of the query sorting to optimize the query evaluation. All the data sources are relevant for the triple pattern tp_3 of the query Q_{LS2}. In addition, the subject, the predicate and the object are variables. Processing this triple pattern first would retrieve all triples in all data data sources and increase the network communication cost. Instead, processing the triple pattern tp_2 first and then using the bindings of the variable *?caff* to filter the results of tp_3 is a better strategy. The bindings processing optimization is dynamically handled during the query evaluation as explained in Section 5.5.1.

- $Q_{LS4_{optimized}} = JOIN\{L \bowtie_{DrugBank} (tp_1, tp_2, tp_3), L \bowtie_{KEGG} (tp_4, tp_5, tp_6)\}$

  $Q_{LS4_{optimized}}$ transformes Q_{LS4} into two local joins to separately evaluate in two data sources whereas the initial query would send 7 triples patterns to data source endpoints.

- $Q_{LS3_{optimized}} = JOIN\{L\bowtie_{DBPedia\ subset}\ (tp_1), D\bowtie_{S_D}\ (tp_2), L\bowtie_{DrugBank}\ (tp_3, tp_4, tp_5)\}$
  with $S_D = (S_{D_{tp2}})\ and\ S_{D_{tp2}} = \{DrugBank, DBPedia\ subset, KEGG\}$

- $Q_{LS5_{optimized}} = JOIN\{L\bowtie_{DrugBank}\ (tp_1, tp_2, tp_4), UNION\{L\bowtie_{S_L}\ (tp_3, tp_5), D\bowtie_{S_D}\ (tp_3, tp_5)\}, L\bowtie_{ChEBI}\ (tp_6)\}$
  with $S_D = (S_{D_{tp3}}, S_{D_{tp5}})\ and\ S_L = S_{D_{tp3}} = S_{D_{tp5}} = \{ChEBI, KEGG\}$

- $Q_{LS6_{optimized}} = JOIN\{JOIN\{L\bowtie_{DrugBank}\ (tp_1, tp_2), L\bowtie_{KEGG}\ (tp_3)\}, UNION\{L\bowtie_{S_L}\ (tp_4, tp_5), D\bowtie_{S_D}\ (tp_4, tp_5)\}\}$
  with $S_D = (S_{D_{tp4}}, S_{D_{tp5}})\ and\ S_L = S_{D_{tp4}} = S_{D_{tp5}} = \{ChEBI, KEGG\}$

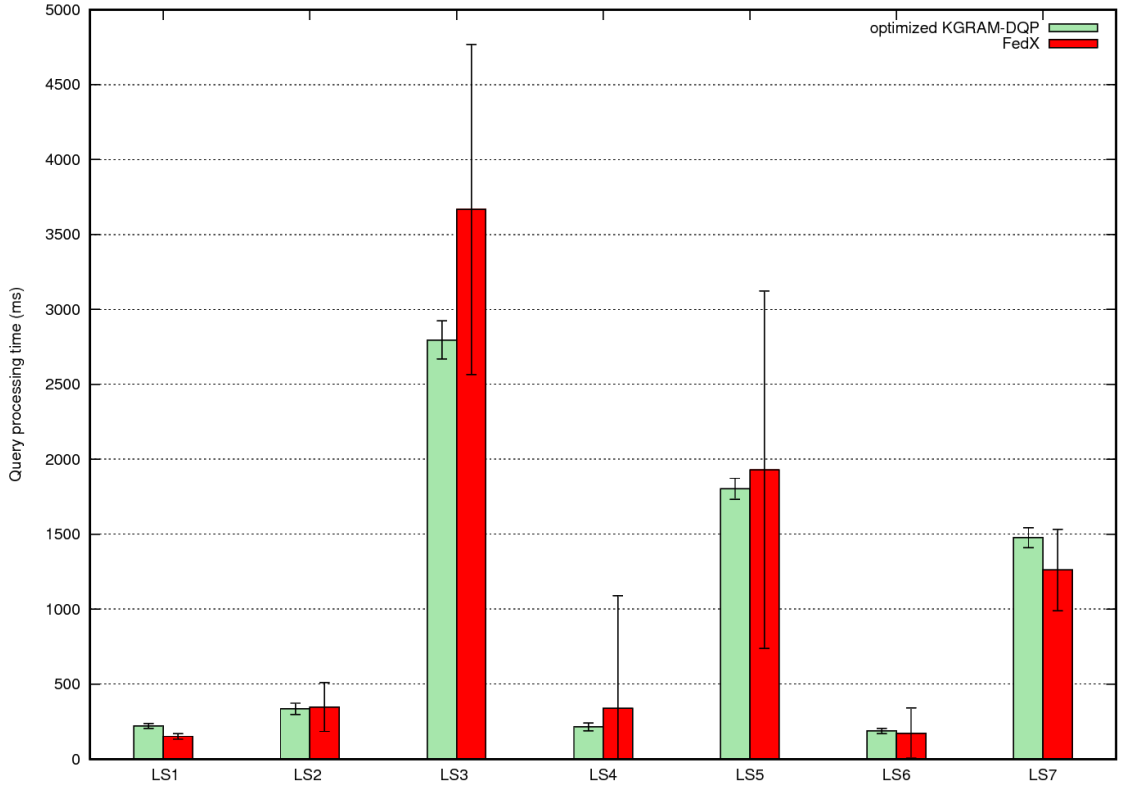
- $Q_{LS7_{optimized}} = JOIN\{L\bowtie_{DrugBank}\ (tp_1, tp_2, OPTIONAL(tp_5)), D\bowtie_{S_D}\ (tp_3), L\bowtie_{DrugBank}\ (tp_4, FILTER(exp))\}$
  with $S_D = (S_{D_{tp3}})\ and\ S_{D_{tp3}} = \{ChEBI, KEGG\}$

$Q_{LS3_{optimized}}$ and $Q_{LS7_{optimized}}$ combine local and distributed joins ($L\bowtie$ and $D\bowtie$). The $D\bowtie$ contains a single triple pattern. Therefore, there is no need to rewrite it as UNION of $D\bowtie$ and $L\bowtie$. Conversely, $Q_{LS5_{optimized}}$ and $Q_{LS6_{optimized}}$ $D\bowtie$ are rewritten in that way as described in Section 5.3. For query Q_{LS7}, the triple patterns tp_1 and tp_2 are associated to the same relevant data source as tp_5 in the *OPTIONAL* clause. The *OPTIONAL* clause also shares variables only with these two triple patterns. For that reason, the *OPTIONAL* clause and the triple patterns p_1 and tp_2 are included in the same local join. This is an optimization of *OPTIONAL* in a straightforward case. A global optimization of *OPTIONAL* clauses would require more investigation due to their complexity [62].

6.3.2.3 Results

Figure 6.3 shows the performance of the Optimized KGRAM-DQP and FedX on the FedBench benchmark. This figure displays the query execution times which are averaged executions of 6 runs and the error bars which show $\pm$ 1 standard deviation. KGRAM-DQP performance is better than FedX performance for Q_{LS3}, Q_{LS4} and Q_{LS5}. FedX query processing time is better for Q_{LS1} and Q_{LS7}. The two engines performances are slightly similar for Q_{LS2} and Q_{LS6}. The latter results can be explained by the fact that FedBench datasets are rather vertically partitioned. Indeed, we are performing the same query rewriting as FedX in the case of vertical partitions. FedX performs *exclusive grouping* which is equivalent to our $L\bowtie$ for vertical partitions. Regarding the other queries, the variation of the two engines performances can be explained by the difference on their query planning optimization strategy. We also note that FedX measures display a large variation whereas the KGRAM-DQP

Figure. 6.3: Optimized KGRAM-DQP - FedX: query processing time



measures display a little variation. Indeed, FedX manages an efficient query results caching which improves the query processing time when queries execution are repeated as in our evaluation method.

Figure 6.4 displays the processing time of FedX without caching (the first run), FedX with caching (the second run) and KGRAM-DQP (first run) for the life science queries (Q_{LS1} to Q_{LS7}). As can be seen, the processing time is greatly reduced between the FedX first and second run. The KGRAM-DQP first evaluation is much more efficient than the FedX first evaluation and sometimes in line with FedX with caching.

The two engines retrieved the same number of results for all queries as shown in Table 6.3. Both KGRAM-DQP and FedX manage to handle the results completeness.

Figure. 6.4: Optimized KGRAM-DQP - FedX (first) - FedX (+caching): query processing time

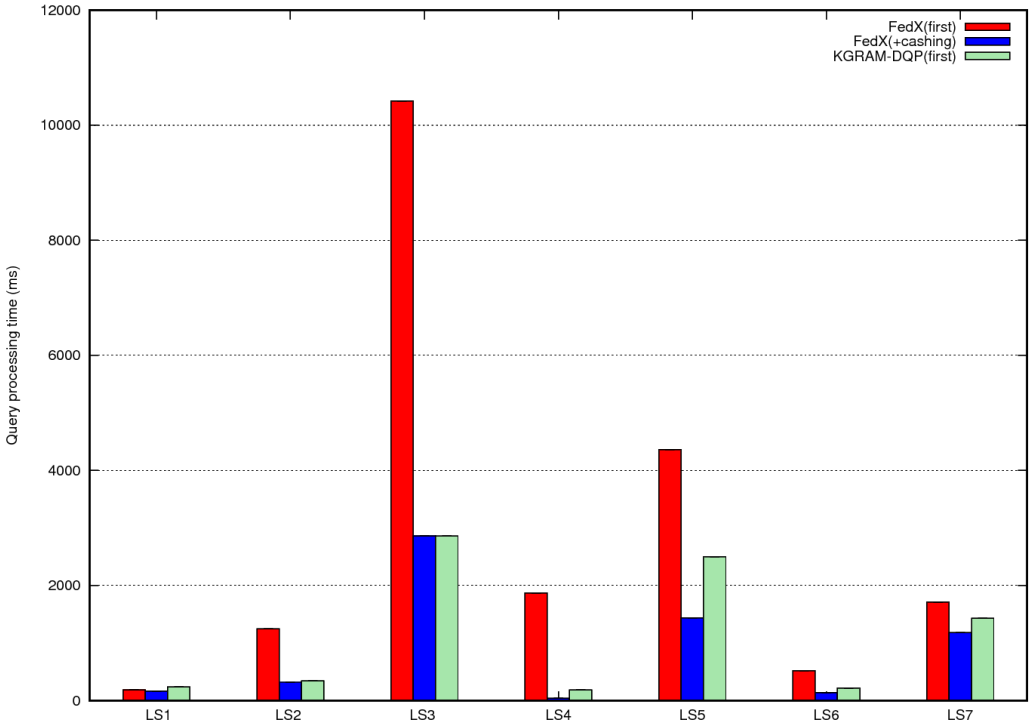


Table. 6.3: Number of results on FedBench

Engines	Queries	Number of results
Optimized KGRAM-DQP / FedX	Q_{LS1}	1159
	Q_{LS2}	333
	Q_{LS3}	9054
	Q_{LS4}	3
	Q_{LS5}	393
	Q_{LS6}	28
	Q_{LS7}	144

With FedBench, the data are connected (i.e. sharing a relation in such a way that the subject is in one source and the object in another one) with always two predicates (*kegg:CompoundId* and *owl:sameAs*) and are more vertically partitioned. Moreover, the queries are simple SELECT query forms. For a more complete comparison, the engines need to be evaluated in a hybrid data partitioning context (horizontal and vertical) with data replication as observed in a real world context and with more diverse SPARQL queries.

6.3.3 KGRAM-DQP vs FedX

This experiment aims at comparing the Optimized KGRAM-DQP with FedX in a hybrid partitioning data context and at underlining the impact of triples replication in data sources over the number of query results. In this experiment we use the same material and methods as in Section 6.3.1.1 (Hybrid Data Partitioning experiment), to compare the results and the number of query results retrieved by the Optimized KGRAM-DQP and the FedX engine (FedX version 3.1). In the remainder, KGRAM-DQP refers to Optimized KGRAM-DQP.

6.3.3.1 Query rewriting and query sorting impact

In this section, we compare the KGRAM-DQP and FedX query rewriting optimizations and their query sorting approaches. FedX performs exclusive grouping optimization for the query rewriting which is similar to our local join operation for vertically partitioned data. However, there is no optimization for horizontally partitioned data in FedX whereas KGRAM-DQP performs the hybrid rewriting described in Section 5.3.

To summarize, in this experiment FedX only generates one local join for demographic data (which is a vertical partition) while KGRAM-DQP also generates local join for geographic data (which is horizontal partition) when possible. Thus, for instance the Query Q_{SELECT} is rewritten by the two engines in P_1 partitioning as follows:

- FedX: $Q_{SELECT_{optimized}}$ = $JOIN\{L \bowtie_{S_1} (tp_4, tp_5), JOIN(tp_1, tp_2, tp_3)\}$

- KGRAM-DQP: $Q_{SELECT_{optimized}}$ = $JOIN\{UNION\{L \bowtie_{S_L} (tp_1, tp_2, tp_3), D \bowtie_{S_D} (tp_1, tp_2, tp_3)\}, L \bowtie_{S_1} (tp_4, tp_5)\}$ $S_D = (S_{D_{tp1}}, S_{D_{tp2}}, S_{D_{tp3}})$ and $S_L = S_{D_{tp1}} = S_{D_{tp2}} = S_{D_{tp3}} = \{S_2, S_3\}$

FedX first evaluates the exclusive group for S_1 $(L \bowtie_{S_1} (tp_4, tp_5))$ then the triple patterns related to data horizontally partitioned $(tp_1, tp_2$ and $tp_3)$ by distributing join over S_2 and S_3. Conversely, KGRAM-DQP first evaluates triple patterns related to data horizontally partitioned then the exclusive group for S_1. In addition, the evaluation of the first part is performed through local joins for S_2 and S_3 and distributed joins between them (our distributed join avoids to recompute the local join results).

Regarding the query planning, FedX gives priority to the *exclusive groups* based on the assumption that they are generally more selective (less data transferred to the engine), while KGRAM-DQP evaluates this exclusive group last thanks to the cost estimation based on the predicate cardinality retrieved during the source selection. As a reminder, the *idxPredicateCardinality* introduced in Section 5.2.1 is depicted below:

- idxPredicateSources = {
  $geo : subdivisionDirecte \rightarrow \{S_1, S_2\}$;
  $geo : codeRegion \rightarrow \{S_1, S_2\}$;
  $geo : nom \rightarrow \{S_1, S_2\}$;
  $demo : population \rightarrow \{S_3\}$;
  $demo : populationTotale \rightarrow \{S_3\}\}$

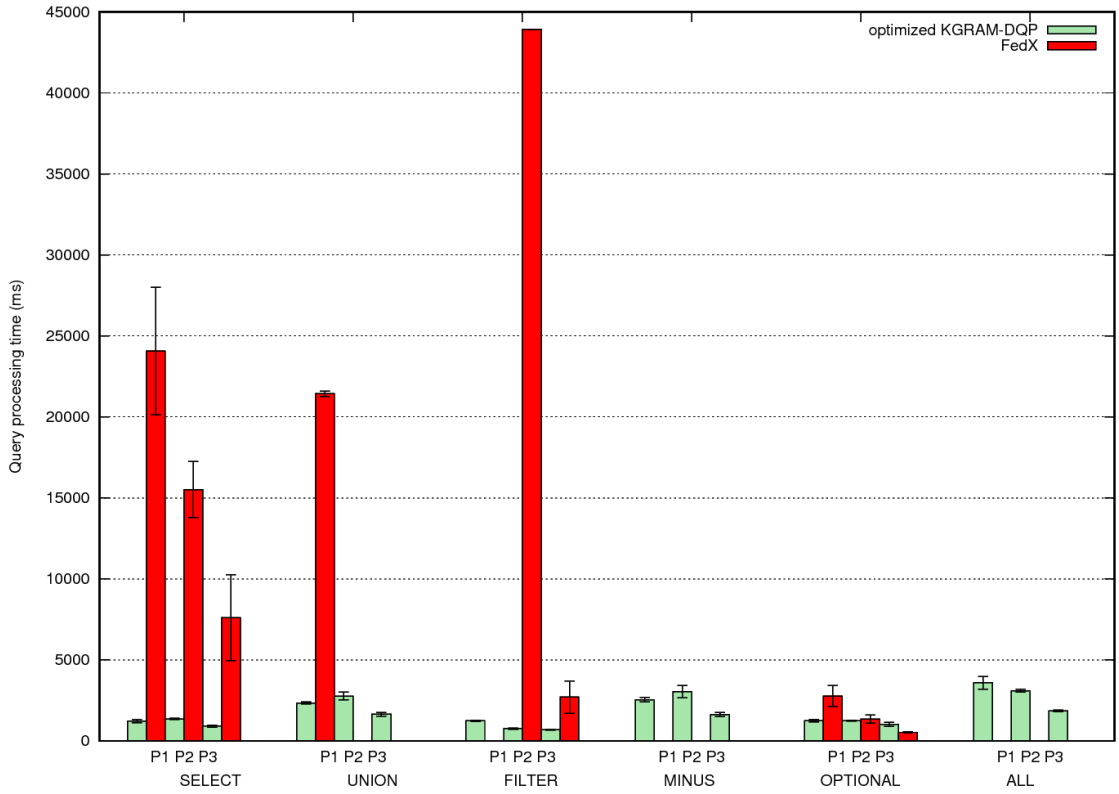
- idxPredicateCardinality = $\{geo : subdivisionDirecte \rightarrow 3934$;
  $geo : codeRegion \rightarrow 27$;
  $geo : nom \rightarrow 41458$;
  $demo : population \rightarrow 37149$;
  $demo : populationTotale \rightarrow 37147\}$

As can be seen, evaluating the exclusive group first will retrieve more triples due to the high cardinality (37149) of the predicates demo:population and demo:populationTotale compared to other predicates, in addition triple patterns subject and object are variables.

6.3.3.2 Query processing performance

Figure 6.5 displays the query processing time for each query using the two engines in the $P_1, P_2\ and\ P_3$ partitions cases. For some queries (*MINUS* and *ALL*) FedX does not provide results. This is explained by the fact that FedX 3.1 does not handle some SPARQL 1.1 features such as *MINUS*. The query *ALL* also contains *MINUS*. The execution of queries *UNION* in $P_1\ and\ P_2$ and *FILTER* in P_1 generate time-out on FedX. Figure 6.5 shows that the Optimized KGRAM-DQP query processing is more efficient than FedX query processing in almost all queries and in all partitions except for *OPTIONAL* in P_3.

Figure. 6.5: Optimized KGRAM-DQP and FedX performance



6.3.3.3 Results

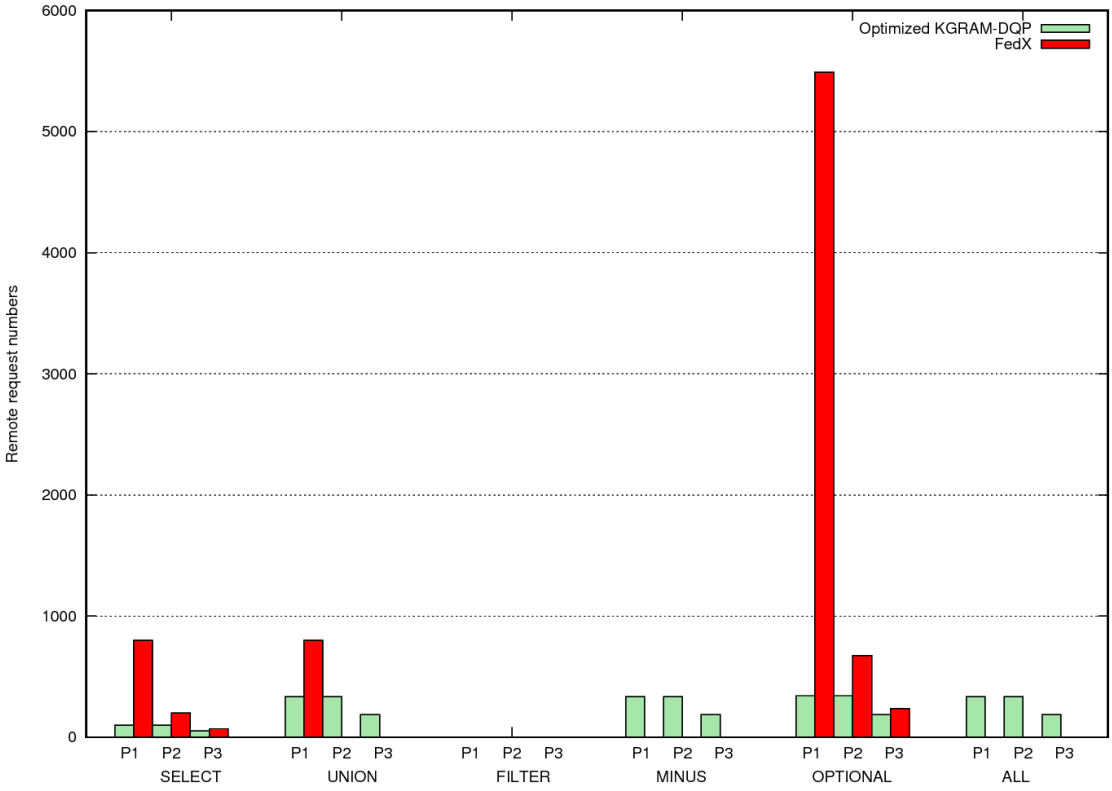
Table 6.4 summarizes the number of results retrieved by the two engines for each query and for the 3 partitioning schemas (P_1, P_2 *and* P_3). As shown in this table, FedX produces more (duplicated) results than the Optimized KGRAM-DQP for most of the queries. For instance, for the query Q_{SELECT}, KGRAM-DQP produces 100 results for P_1 whereas FedX generates 800 results. This is explained by the fact that FedX does not handle triples replication on data sources, thus generating duplicated results and making the query processing less efficient. X indicates the queries which cannot be processed due to the fact that FedX (version 3.1) is not compliant with SPARQL 1.1 and T indicates the queries that timeouted.

Figure 6.6 also illustrates the difference on the number of results between the Optimized KGRAM-DQP and FedX.

Table. 6.4: Number of results

Engines	Queries	P_1 **partitioning**	P_2 **partitioning**	P_3 partitioning
Optimized KGRAM-DQP	Q_{SELECT}	100	100	54
	Q_{UNION}	335	335	185
	Q_{MINUS}	335	335	185
	Q_{FILTER}	1	1	0
	$Q_{OPTIONAL}$	343	343	186
	Q_{ALL}	395	395	11
FedX	Q_{SELECT}	800	199	70
	Q_{UNION}	800	T	T
	Q_{MINUS}	X	X	X
	Q_{FILTER}	T	2	0
	$Q_{OPTIONAL}$	5488	673	237
	Q_{ALL}	X	X	X

Figure. 6.6: Number of results retrieved on INSEEE data



6.4 Conclusions

This chapter introduces the KGRAM-DQP engine, which implements our federated query processing approach within the Corese semantic Web framework. We propose 3 experiments to assess the impact of our optimization strategies regarding query processing efficiency, result completeness and data replication challenges. A first experimental study simulates both vertical and horizontal partitioning of data context in several kinds of partition. Based on this experiment we compared the basic KGRAM-DQP engine with an optimized version in which we implemented the static and dynamic optimization proposed in Chapter 5. The results show a significant performance improvement regarding the query processing time and the number of remote requests. Afterwards, we compared the optimized KGRAM-DQP with FedX on FedBench [73] benchmark. In this experiment, KGRAM-DQP results are consistent with FedX results about the number of results and their completeness. KGRAM-DQP query processing performance is most of the time better than FedX performance for first evaluation and sometimes in line with FedX with query results caching for the following evaluation. KGRAM-DQP results also display more stability than FedX results. Finally, we compared the engines in the hybrid data partition context through the first experimental setup. In this one, KGRAM-DQP is more efficient than FedX. Moreover, KGRAM-DQP managed to handle data replication and more expressive SPARQL queries. In summary, we can that our federated engine manages to efficiently query distributed data sources in any kind of data partitioning scheme through our optimization strategies while addressing results completeness, results redundancy side effect of triple replication and query expressiveness challenges.

7 Conclusions and Perspectives

Contents

7.1 Summary 102
7.2 Perspectives 104

Driven by the Semantic Web standards, more and more RDF data sources are made available and connected over the Web by data providers, leading to a large distributed network. However, processing these data sources becomes very challenging for data consumers due to data distribution and their volume growth. The Federated query processing approach allows querying distributed data sources by relying on Distributed Query Processing (DQP) techniques. A naive implementation of DQP may generate a tremendous number of remote requests towards data sources and numerous intermediate results, thus leading to costly network communications. Furthermore, the distributed query semantics is often overlooked. Query expressiveness, data partitioning, and data replication are other challenges to be taken into account in order to implement sound and efficient distributed queries.

7.1 Summary

In Chapter 2 we presented the general principles of data integration through mappings, taking into account data fragmentation. Then we focused on Linked Open Data integration which is the equivalent of distributed databases integration at the Semantic Web scale. Indeed, the Web of Data is a network of distributed, autonomous and linked data sources. We described the Linked Open Data integration principles, data representation and querying. LOD integration mainly relies on W3C standards, namely RDF and SPARQL. The continuously increasing number of RDF datasets made available by data providers lead to complex information and queries to process. Distributed Query Processing (DQP) introduced in Chapter 3 is the main approach used to perform query processing in this context. DQP, also called federated query processing, is usually decomposed in four main steps: (i) source selection, (ii) query rewriting, (iii) query planning and (iv) query evaluation. We reviewed several state-of-the-art DQP approaches and analyzed them through the steps identified beforehand. Based on the optimization strategies analysis, we highlighted several challenges to address, such as source selection accuracy, query results completeness, data replication, SPARQL distributed query semantics and query expressiveness, for a more efficient, reliable and scalable federated query processing.

In Chapter 4, we tackled the first objective of this thesis which is to propose a SPARQL and RDF compliant Distributed Query Processing semantics in order to ensure the query semantics and expressiveness preservation during federated query evaluation. We first showed the need for a semantics to avoid query results disparity between query engines and to provide more reliability in the results produced. Then, we proposed a federated query semantics on top of W3C standards while addressing named graph collision, blank nodes and triples replication constraints in a distributed

and autonomous data sources context. Following this, we specified the semantics of federated SPARQL queries on top of the standard SPARQL semantics through a set of rewrite rules relying on service clauses.

In Chapter 5, we introduced our contribution regarding the second objective of this thesis, which is to transparently and efficiently address autonomous and distributed data sources. Thus, we proposed both static and dynamic optimizations to enhance the federated query processing performance. Static optimizations are performed during query rewriting and query planning steps while dynamic optimizations are processed during query evaluation. Our static optimizations rely on the result of the source selection step in which we proposed a Sampling Query-Based approach. This approach builds two indices. The source selection index identifies relevant data sources and collects information on data distribution, while predicates index gives information on predicates cardinality. Using the source selection index, we proposed a Hybrid BGP-Triple query rewriting approach which addresses both horizontal and vertical data partitions and reduces the query engine workload. Using the predicates index, we proposed a static query sorting approach which combines cost estimation, heuristics on query patterns and query expressions links. Our dynamic optimizations are achieved during the query evaluation step at three levels. First, the bindings (already known values of variables) are propagated to the following sub-queries sharing the same variables. Secondly, we use parallelism to concurrently query remote data sources and independent remote requests. Finally, we use triple results duplicate-aware evaluation to avoid results redundancy which are side effects of triples replication.

Finally, in Chapter 6, we assessed our federated query processing strategies regarding query evaluation efficiency, results completeness and data replication challenges. We implemented our approach in the KGRAM-DQP query engine. We ran several experiments which showed an improvement of the KGRAM-DQP performance while handling results completeness, query expressiveness, and results redundancy caused by triple replication. We also compared KGRAM-DQP with FedX in two data partitioning contexts. In vertical partitions, without triple replication, KGRAM-DQP results are consistent with FedX results and its performance is better in some cases. However, we have also observed that a results caching system like FedX can improve the query processing performance when queries evaluation is repeated. In a Hybrid partition context (both vertical and horizontal partitions), KGRAM-DQP is more efficient than FedX. Unlike FedX, KGRAM-DQP handles replication and manages more expressive queries.

In summary, we proposed in this thesis a transparent and trustworthy federated query processing approach without any prior knowledge on distributed data sources. In this respect, we first introduced a SPARQL and RDF-compliant Distributed Query

Processing semantics which preserves the SPARQL language expressiveness. This semantics also overcomes the query results disparity issue between the federated query engines and thus increases their reliability. Afterwards, we proposed static and dynamic optimization strategies for federated query processing steps while addressing data partitioning, query results completeness, data replication, and query processing performance. Our strategies reduce the number of data sources requested, decrease the query engine workload through triple patterns grouping for both horizontal and vertical data partitions, and efficiently perform a duplicate-aware query evaluation.

However, some limitations of this work can be highlighted. Our source selection approach is individually achieved on each predicate for each data source and thus may generate many remote requests. The triple patterns grouping approach relies on heuristics, in particular by considering whether the whole BGP is connected or not, which can restrict the number of BGPs generated. These heuristics may be extended by considering the subsets of non-connected BGPs when their triple patterns are connected. In addition, our query sorting approach generates an optimized query plan performed at the query execution step as it stands. It would be interesting to make this plan more adaptive during the query execution. Finally, the optimization of some SPARQL features such as FILTER EXISTS in distributed queries could be investigated in-depth in order to reap the full benefits of SPARQL expressiveness in a reasonable query processing performance.

Through this work, we have seen that addressing distributed data sources querying through a federated query processing approach implies coping with several challenges and thus might be tedious. Properly overcoming all these challenges can be time consuming, thus a trade-off needs to be made between the optimization accuracy and query processing time. Moreover, federated query engines should have an adaptive approach rather than trying to design a one-size-fits-all solution. Indeed, in some cases a choice needs to be made between some challenges. For instance, it is often not possible to support the query results completeness, and at the same quickly retrieving the results for large data source. In this sense, allowing end users to specify their requirements might help federated query engines to adapt their processing strategies and enhance their performance.

7.2 Perspectives

The work done during this thesis can be pursued and improved in many aspects to overcome the highlighted limitations and towards a more flexible federated query processing approach. In that respect the following perspectives can be considered:

- Query processing performance: Regarding the query execution efficiency, we first introduced a Sampling Query-Based approach to acquire knowledge (statistics) on data sources and to identify relevant data sources for predicates of federated queries. Indeed, in Chapter 6, we used a sampling query for each predicate and sent it to all remote data sources. In very short term, this sampling phase can be improved by using SPARQL BIND clauses to combine all the predicates in one query for each data source and to assign the result for each predicate to a variable. This would allow reducing the number of remote requests to data sources and processing time during the source selection step. As a future work on the longer term, this sampling phase can also be improved by taking into account available information on links between data sources entities and vocabularies. Indeed, the accuracy of source selection can be adversely affected by the use of popular properties, and the increasing number of data sources involved has a negative influence on query processing performance [66]. Several approaches [70, 17] have been proposed to address this issue and showed promising results with FedX. Combining these different approaches and using the most suitable one to the query federation context, will enhance the relevant sources accuracy while ensuring data sources catalogs freshness at the same time.

  The Query planning approach can also be improved by using Query Performance Prediction (QPP) [40]. In this approach, machine learning techniques are used to predict the query processing performance by learning from the execution time of already processed queries. In a real-world context, in which queries can be similar or repeated, using this approach can generate more optimal query plans if the prediction tasks are achieved within a reasonable time.

  As shown in Chapter 6, during the query evaluation, the use of caching can improve the query processing performance. A Graph-Aware workload-adaptive caching approach [60] has also been proposed for SPARQL queries and showed encouraging performance over large scale RDF datasets. In our work, we generated our optimized query plan using knowledge retrieved during source selection, and we executed this plan using bindings and parallelism. However, adjusting the initial plan during the execution, may be required due to the data sources response time and the number of intermediate results gathered. Thus, several adaptive query optimization approaches have been proposed [23, 2, 55, 56] for federated query processing. After rewriting the federated query as unions of *SERVICE* clauses as described in the Chapter 4, instead of letting the query engine send them one by one to the endpoints, it could delegate this processing to the endpoints. Indeed, with the whole query and the relevant data sources URLs, each endpoint, after executing its sub-query,

could ask the next endpoint through its URL to execute the following subquery. Thus, the remote requests and intermediate results processing can be distributed between the endpoints and the engine and perhaps enhance the query processing performance. But this requires a controlled execution environment with reliable endpoints, especially regarding query processing performance. In our case, where KGRAM can be used as both engine and endpoint, this might be considered in order to further reduce the engine workload.

- Query expressiveness: On one hand, we proposed a semantics for federated queries and specified rewriting for SPARQL. On the other hand, we implemented our federated query engine on top of the KGRAM engine which is fully compliant with SPARQL. Moreover, to deal with performance issues related to the complexity of SPARQL [62, 74], we applied several optimizations such as pushing relevant FILTERs. As a follow-up to this work, it would be interesting to address more complex FILTERs expressions issues such as FILTER EXISTS and also others SPARQL features optimization such as NAMED GRAPH in order to find new heuristics to increase the distributed SPARQL query efficiency.

- Data sources heterogeneity : In this work we focused on homogeneous data. However, in real-world Web scale applications, the data sources to integrate can be in different formats such as SQL or NoSQL databases. To allow data consumers to access to more knowledge bases, the R2RML [65] mapping language was proposed for relational databases. This language was extended to manage NoSQL databases (xR2RML [52]). The SPARQL Federated query engine can take benefit from this works to manage heterogeneous data sources.

Bibliography

[1]Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. „Scalable Semantic Web Data Management Using Vertical Partitioning“. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 411–422 (cit. on p. 12).

[2]Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. „ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints“. In: *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*. ISWC'11. Bonn, Germany: Springer-Verlag, 2011, pp. 18–34 (cit. on pp. 24, 26, 33, 52, 105).

[3]Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. „Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design“. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: ACM, 2004, pp. 359–370 (cit. on p. 12).

[4]Ziya Akar, Tayfun Gökmen Halaç, Erdem Eser Ekinci, and Oguz Dikenelli. „Querying the Web of Interlinked Datasets using VOID Descriptions.“ In: *LDOW*. Ed. by Christian Bizer, Tom Heath, Tim Berners-Lee, and Michael Hausenblas. Vol. 937. CEUR Workshop Proceedings. CEUR-WS.org, 2012 (cit. on pp. 24, 27, 52).

[5]Keith Alexander and Michael Hausenblas. „Describing linked datasets - on the design and usage of void, the vocabulary of interlinked datasets“. In: *Linked Data on the Web Workshop (LDOW 09)*. Madrid, Spain, Apr. 2009 (cit. on pp. 25, 26).

[6]Medha Atre. „Left Bit Right: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins)“. In: *SIGMOD Conference*. ACM, 2015, pp. 1793–1808 (cit. on p. 61).

[7]Sören Auer, Christian Bizer, Georgi Kobilarov, et al. „DBpedia: A Nucleus for a Web of Open Data“. In: *The Semantic Web*. Ed. by Karl Aberer, Key-Sun Choi, Natasha Noy, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 722–735 (cit. on p. 19).

[8]T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. United States, 1998 (cit. on pp. 2, 13).

[9]Tim Berners-Lee. *Information Management: A Proposal*. en. 1989 (cit. on p. 2).

[10]Tim Berners-Lee. *Linked Data, in Design Issues of the WWW* (cit. on pp. 2, 13).

[11]Tim Berners-Lee, James Hendler, and Ora Lassila. „The Semantic Web“. In: *Scientific American* 284.5 (May 2001), pp. 34–43 (cit. on p. 2).

[12]C. Bizer. „The Emerging Web of Linked Data“. In: *IEEE Intelligent Systems* 24.5 (2009), pp. 87–92 (cit. on p. 13).

[13]C. Bizer, T. Heath, and T. Berners-Lee. „Linked data - the story so far“. In: *Int. J. Semantic Web Inf. Syst.* 5.3 (2009), 1–22 (cit. on p. 2).

[14]Carlos Buil-Aranda, Axel Polleres, and Jürgen Umbrich. „Strategies for Executing Federated Queries in SPARQL1.1“. In: *The Semantic Web – ISWC 2014*. Ed. by Peter Mika, Tania Tudorache, Abraham Bernstein, et al. Cham: Springer International Publishing, 2014, pp. 390–405 (cit. on p. 72).

[15]Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. „On the Expressive Power of Data Integration Systems“. In: *Conceptual Modeling — ER 2002*. Ed. by Stefano Spaccapietra, Salvatore T. March, and Yahiko Kambayashi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003 (cit. on p. 9).

[16]Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. „What Is Query Rewriting?“ In: *Cooperative Information Agents IV - The Future of Information Agents in Cyberspace*. Ed. by Matthias Klusch and Larry Kerschberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 51–59 (cit. on p. 22).

[17]Ethem Cem Ozkan, Muhammad Saleem, Erdogan Dogdu, and Axel-Cyrille Ngonga Ngomo. „UPSP: Unique Predicate-based Source Selection for SPARQL Endpoint Federation“. In: *PROFILES at Extended Semantic Web Conference (ESWC)*. 2016 (cit. on p. 105).

[18]Sharma Chakravarthy, Jaykumar Muthuraj, Ravi Varadarajan, and Shamkant B. Navathe. „An Objective Function for Vertically Partitioning Relations in Distributed Databases and Its Analysis“. In: *Distrib. Parallel Databases* 2.2 (Apr. 1994), pp. 183–207 (cit. on p. 12).

[19]O. Corby, R. Dieng-Kuntz, F. Gandon, and C. Faron-Zucker. „Searching the semantic Web: approximate query processing based on ontologies“. In: vol. 21. 2006, pp. 20–27 (cit. on p. 80).

[20]Olivier Corby and Catherine Faron Zucker. „The KGRAM Abstract Machine for Knowledge Graph Querying“. In: *Web Intelligence and Intelligent Agent Technology*. Toronto, Canada, Aug. 2010, pp. 338 –341 (cit. on p. 80).

[21]Olivier Corby, Alban Gaignard, Catherine Faron-Zucker, and Johan Montagnat. „KGRAM Versatile Inference and Query Engine for the Web of Linked Data“. In: *IEEE/WIC/ACM International Conference on Web Intelligence(WI'12)*. Macao, China, Dec. 2012 (cit. on p. 80).

[22]Olivier Corby, Rose Dieng-Kuntz, and Catherine Faron Zucker. „Querying the Semantic Web with Corese Search Engine“. In: *European Conference on Artificial Intelligence*. Valence, Spain, Aug. 2004 (cit. on p. 80).

[23]Amol Deshpande, Zachary Ives, and Vijayshankar Raman. „Adaptive Query Processing“. In: *Found. Trends databases* 1.1 (Jan. 2007), pp. 1–140 (cit. on pp. 33, 105).

[24]Aleksandar Dimovski, Goran Velinov, and Dragan Sahpaski. „Horizontal Partitioning by Predicate Abstraction and Its Application to Data Warehouse Design“. In: *Proceedings of the 14th East European Conference on Advances in Databases and Information Systems*. ADBIS'10. Novi Sad, Serbia: Springer-Verlag, 2010, pp. 164–175 (cit. on p. 12).

[25]Orri Erling and Ivan Mikhailov. „RDF Support in the Virtuoso DBMS“. In: *Networked Knowledge - Networked Media: Integrating Knowledge Management, New Media Technologies and Semantic Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 7–24 (cit. on p. 89).

[26]Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. „Query Optimization in the Presence of Limited Access Patterns“. In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. SIGMOD '99. Philadelphia, Pennsylvania, USA: ACM, 1999, pp. 311–322 (cit. on p. 33).

[27]*French demographic*. Accessed: 2018-09-01. 2014 (cit. on p. 54).

[28]*French geographic*. Accessed: 2018-09-01. 2014 (cit. on p. 54).

[29]Marc Friedman, Alon Levy, and Todd Millstein. „Navigational Plans for Data Integration“. In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*. AAAI '99/IAAI '99. Orlando, Florida, USA: American Association for Artificial Intelligence, 1999, pp. 67–73 (cit. on p. 11).

[30]Alban Gaignard. „Distributed knowledge sharing and production through collaborative e-Science platforms“. Theses. Université Nice Sophia Antipolis, Mar. 2013 (cit. on pp. 4, 89).

[31]Olaf Görlitz and Steffen Staab. „Federated Data Management and Query Optimization for Linked Open Data“. In: *New Directions in Web Data Management 1*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 109–137 (cit. on p. 3).

[32]Olaf Görlitz and Steffen Staab. „SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions“. In: *Conference on Consuming Linked Data - Volume 782*. COLD'11. Bonn, Germany, 2010, pp. 13–24 (cit. on pp. 24, 26, 52).

[33]Ashish Gupta and Inderpal Singh Mumick. „Materialized Views“. In: ed. by Ashish Gupta and Iderpal Singh Mumick. Cambridge, MA, USA: MIT Press, 1999. Chap. Maintenance of Materialized Views: Problems, Techniques, and Applications, pp. 145–157 (cit. on p. 9).

[34]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. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 276–285 (cit. on pp. 8, 32).

[35]Peter J. Haas and Joseph M. Hellerstein. „Ripple Joins for Online Aggregation“. In: *SIGMOD Rec.* 28.2 (June 1999), pp. 287–298 (cit. on p. 9).

[36]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. Graz, Austria: ACM, 2010, 5:1–5:9 (cit. on p. 8).

[37]Alon Y. Halevy. „Answering queries using views: A survey“. In: *The VLDB Journal* 10.4 (2001), pp. 270–294 (cit. on p. 10).

[38]Andreas Harth and Sebastian Speiser. „On Completeness Classes for Query Evaluation on Linked Data“. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI'12. Toronto, Ontario, Canada: AAAI Press, 2012, pp. 613–619 (cit. on p. 12).

[39]Olaf Hartig, Ian Letter, and Jorge Pérez. „A Formal Framework for Comparing Linked Data Fragments“. In: *The Semantic Web – ISWC 2017*. Ed. by Claudia d'Amato, Miriam Fernandez, Valentina Tamma, et al. Cham: Springer International Publishing, 2017, pp. 364–382 (cit. on p. 3).

[40] Rakebul Hasan. „Predicting query performance and explaining results to assist Linked Data consumption“. Theses. Université Nice Sophia Antipolis, Nov. 2014 (cit. on p. 105).

[41] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011 (cit. on p. 13).

[42] 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. Scottsdale, Arizona: ACM, 2012, 2:1–2:8 (cit. on p. 26).

[43] 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. Scottsdale, Arizona: ACM, 2012, 2:1–2:8 (cit. on p. 35).

[44] Jiewen Huang, Daniel J. Abadi, and Kun Ren. „Scalable SPARQL Querying of Large RDF Graphs“. In: *PVLDB* 4 (2011), pp. 1123–1134 (cit. on p. 19).

[45] Toshihide Ibaraki and Tiko Kameda. „On the Optimal Nesting Order for Computing N-relational Joins“. In: *ACM Trans. Database Syst.* 9.3 (Sept. 1984), pp. 482–502 (cit. on p. 29).

[46] Günter Ladwig and Thanh Tran. „Linked Data Query Processing Strategies“. In: *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*. ISWC'10. Shanghai, China: Springer-Verlag, 2010, pp. 453–469 (cit. on p. 3).

[47] Jens Lehmann, Robert Isele, Max Jakob, et al. „DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia“. In: *Semantic Web* 6 (2015), pp. 167–195 (cit. on p. 19).

[48] Maurizio Lenzerini. „Data Integration: A Theoretical Perspective“. In: *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '02. Madison, Wisconsin: ACM, 2002, pp. 233–246 (cit. on p. 10).

[49] Alon Y. Levy. „Logic-Based Techniques in Data Integration“. In: *Logic-Based Artificial Intelligence*. Boston, MA: Springer US, 2000, pp. 575–595 (cit. on p. 10).

[50] *LODStats: a statement-stream-based approach for gathering comprehensive statistics about RDF datasets*. Accessed: 2018-09-01 (cit. on p. 25).

[51] Steven "Lynden, Isao Kojima, Akiyoshi Matono, et al. „ADERIS: An Adaptive Query Processor for Joining Federated SPARQL Endpoints“. In: *On the Move to Meaningful Internet Systems: OTM 2011: Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17-21, 2011, Proceedings, Part II*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 808–817 (cit. on pp. 24, 26, 52).

[52] Franck Michel. „Integrating Heterogeneous Data Sources in the Web of Data“. Theses. Université Côte d'Azur, Mar. 2017 (cit. on pp. 11, 106).

[53] Gabriela Montoya, Maria-Esther Vidal, Oscar Corcho, Edna Ruckhaus, and Carlos Buil-Aranda. „Benchmarking Federated SPARQL Query Engines: Are Existing Testbeds Enough?“ In: *The Semantic Web – ISWC 2012*. Ed. by Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 313–324 (cit. on p. 42).

[54] Gabriela Montoya, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. „Fedra: Query Processing for SPARQL Federations with Divergence“. In: May 2014 (cit. on p. 35).

[55]Damla Oguz, Shaoyi Yin, Abdelkader Hameurlain, Belgin Ergenç, and Oguz Dikenelli. „Adaptive Join Operator for Federated Queries over Linked Data Endpoints“. In: *20th East-European Conference on Advances in Databases and Information Systems (ADBIS 2016)*. Vol. 9809. Prague, Czech Republic, Aug. 2016, pp. 275–290 (cit. on p. 105).

[56]Damla Oguz, Shaoyi Yin, Belgin Ergenç, Abdelkader Hameurlain, and Oguz Dikenelli. „Extended Adaptive Join Operator with Bind-Bloom Join for Federated SPARQL Queries“. In: *Int. J. Data Warehous. Min.* 13.3 (July 2017), pp. 47–72 (cit. on p. 105).

[57]*OWL Web Ontology Language Overview. W3C Recommendation*. Accessed: 2018-09-01 (cit. on p. 2).

[58]Tamer M. Özsu and Patrick Valduriez. „Principles of Distributed Database Systems, third edition“. In: Springer, 2011, p. 845 (cit. on pp. 9, 22, 40).

[59]HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. „Verifying Completeness of Relational Query Results in Data Publishing“. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pp. 407–418 (cit. on p. 12).

[60]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. Melbourne, Victoria, Australia: ACM, 2015, pp. 1777–1792 (cit. on p. 105).

[61]Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. „Semantics and Complexity of SPARQL“. In: *ACM Transactions on Database Systems* 34.3 (Aug. 2009) (cit. on p. 61).

[62]Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. „Semantics and Complexity of SPARQL“. In: *ACM Trans. Database Syst.* 34.3 (Sept. 2009), 16:1–16:45 (cit. on pp. 93, 106).

[63]Bastian Quilitz and Ulf Leser. „Querying Distributed RDF Data Sources with SPARQL“. In: *The Semantic Web: Research and Applications*. Ed. by Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 524–538 (cit. on p. 23).

[64]Bastian Quilitz and Ulf Leser. „Querying Distributed RDF Data Sources with SPARQL“. In: *European Semantic Web Conference on The Semantic Web: Research and Applications*. ESWC'08. Tenerife, Canary Islands, Spain: Springer-Verlag, 2008, pp. 524–538 (cit. on pp. 24, 26, 52).

[65]*R2RML: RDB to RDF Mapping Language. W3C Recommendation 2012*. Accessed: 2018-09-01. 2012 (cit. on p. 106).

[66]N. A. Rakhmawati and M. Hausenblas. „On the Impact of Data Distribution in Federated SPARQL Queries“. In: *2012 IEEE Sixth International Conference on Semantic Computing*. 2012, pp. 255–260 (cit. on p. 105).

[67]*RDF 1.1 Semantics. W3C Recommendation 25 February 2014*. Accessed: 2018-09-01. 2014 (cit. on pp. 38, 39).

[68]*RDF 1.1 XML Syntax, W3C Recommendation 25 February 2014*. Accessed: 2018-09-01. 2014 (cit. on pp. 2, 14).

[69]*RDF Schema 1.1, W3C Recommendation 25 February 2014*. Accessed: 2018-09-01. 2014 (cit. on p. 2).

[70] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. „HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation“. In: *The Semantic Web: Trends and Challenges*. Ed. by Valentina Presutti, Claudia d'Amato, Fabien Gandon, et al. Cham: Springer International Publishing, 2014, pp. 176–191 (cit. on p. 105).

[71] Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. „A Fine-Grained Evaluation of SPARQL Endpoint Federation Systems“. In: *Semantic Web Journal* (2014) (cit. on p. 89).

[72] Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, Josiane Xavier Parreira, Helena F. Deus, and Manfred Hauswirth. „DAW: Duplicate-AWare Federated Query Processing over the Web of Data“. In: *The Semantic Web – ISWC 2013*. Ed. by Harith Alani, Lalana Kagal, Achille Fokoue, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 574–590 (cit. on p. 35).

[73] Michael Schmidt, Olaf Görlitz, Peter Haase, et al. „FedBench: A Benchmark Suite for Federated Semantic Data Query Processing“. In: *The Semantic Web – ISWC 2011*. Ed. by Lora Aroyo, Chris Welty, Harith Alani, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 585–600 (cit. on pp. 89, 100).

[74] Michael Schmidt, Michael Meier, and Georg Lausen. „Foundations of SPARQL Query Optimization“. In: *Proceedings of the 13th International Conference on Database Theory*. ICDT '10. Lausanne, Switzerland: ACM, 2010, pp. 4–33 (cit. on p. 106).

[75] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. „Fedx: optimization techniques for federated query processing on linked data“. In: *international Semantic Web conference(ISWC'11)*. Bonn, Germany, Oct. 2011, pp. 601–616 (cit. on pp. 24, 27, 52).

[76] 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. Boston, Massachusetts: ACM, 1979, pp. 23–34 (cit. on p. 29).

[77] *Semantic Web: standards*. Accessed: 2018-09-01 (cit. on p. 2).

[78] *SPARQL 1.1 Overview, W3C Recommendation 21 March 2013*. Accessed: 2018-09-01 (cit. on pp. 2, 16).

[79] *SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013*. Accessed: 2018-09-01. 2013 (cit. on p. 41).

[80] *SPARQL 1.1 Service Description. W3C Recommendation 21 March 2011*. Accessed: 2018-09-01. 2011 (cit. on p. 25).

[81] *SPARQL Endpoints Status : Datahub.io*. Accessed: 2018-09-01 (cit. on pp. 19, 25).

[82] Markus Stocker and Michael Smith. „Owlgres: A scalable OWL reasoner“. In: *CEUR Workshop Proceedings*. Vol. 432. Oct. 2008 (cit. on p. 30).

[83] 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 '08. Beijing, China: ACM, 2008, pp. 595–604 (cit. on p. 28).

[84]Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. „Heuristics-based Query Optimisation for SPARQL“. In: *Proceedings of the 15th International Conference on Extending Database Technology*. EDBT '12. Berlin, Germany: ACM, 2012, pp. 324–335 (cit. on p. 34).

[85]Tolga Urhan and Michael J. Franklin. „XJoin: A Reactively-Scheduled Pipelined Join Operator“. In: *IEEE Data Eng. Bull.* 23 (2000), pp. 27–33 (cit. on p. 33).

[86]*URIs, URLs, and URNs: Clarifications and Recommendations 1.0*. Accessed: 2018-09-01. 2001 (cit. on p. 13).

[87]Ruben Verborgh, Miel Vander Sande, Olaf Hartig, et al. „Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web“. In: *Journal of Web Semantics* 37–38 (Mar. 2016), pp. 184–206 (cit. on p. 3).

[88]Ruben Verborgh, Miel Vander Sande, Pieter Colpaert, et al. „Web-Scale Querying through Linked Data Fragments“. In: *Proceedings of the 7th Workshop on Linked Data on the Web*. Ed. by Christian Bizer, Tom Heath, Sören Auer, and Tim Berners-Lee. Vol. 1184. CEUR Workshop Proceedings. Apr. 2014 (cit. on p. 3).

[89]Xin Wang, Thanassis Tiropanis, and Hugh Davis. „LHD: Optimising linked data query processing using parallelisation“. In: *CEUR Workshop Proceedings*. Vol. 996. May 2013 (cit. on pp. 24, 26, 52).

Abstract

Driven by the Semantic Web standards, an increasing number of RDF data sources are published and connected over the Web by data providers, leading to a large distributed linked data network. However, exploiting the wealth of these data sources is very challenging for data consumers considering the data distribution, their volume growth and data sources autonomy. In the Linked Data context, federation engines allow querying these distributed data sources by relying on Distributed Query Processing (DQP) techniques. Nevertheless, a naive implementation of the DQP approach may generate a tremendous number of remote requests towards data sources and numerous intermediate results, thus leading to costly network communications. Furthermore, the distributed query semantics is often overlooked. Query expressiveness, data partitioning, and data replication are other challenges to be taken into account. To address these challenges, we first proposed in this thesis a SPARQL and RDF compliant Distributed Query Processing semantics which preserves the SPARQL language expressiveness. Afterwards, we presented several strategies for a federated query engine that transparently addresses distributed data sources, while managing data partitioning, query results completeness, data replication, and query processing performance. We implemented and evaluated our approach and optimization strategies in a federated query engine to prove their effectiveness.

Keywords: Semantic Web, Web of Data, Linked Data, Linked Open Data, Data Integration, Distributed Query Processing, Federated query evaluation, SPARQL, Query Optimization

Résumé

De plus en plus de sources de données liées sont publiées à travers le Web en s'appuyant sur les technologies du Web sémantique, formant ainsi un large réseau de données distribuées. Cependant il est difficile pour les consommateurs de données de profiter de la richesse de ces données, compte tenu de leur distribution, de l'augmentation de leur volume et de l'autonomie des sources de données. Les moteurs fédérateurs de données permettent d'interroger ces sources de données en utilisant des techniques de traitement de requêtes distribuées. Cependant, une mise en œuvre naïve de ces techniques peut générer un nombre considérable de requêtes distantes et de nombreux résultats intermédiaires entraînant ainsi un long temps de traitement des requêtes et des communications réseau coûteuse. Par ailleurs, la sémantique des requêtes distribuées est souvent ignorée. L'expressivité des requêtes, le partitionnement des données et leur réplication sont d'autres défis auxquels doivent faire face les moteurs de requêtes. Pour répondre à ces défis, nous avons d'abord proposé une sémantique des requêtes distribuées compatible avec les standards SPARQL et RDF qui préserve l'expressivité de SPARQL. Nous avons ensuite présenté plusieurs stratégies d'optimisation pour un moteur de requêtes fédérées qui interroge de manière transparente des sources de données distribuées. La performance de ces optimisations est évaluée sur une implémentation d'un moteur de requêtes distribuées SPARQL.

Mots-clés: Web sémantique, Web de Données, Données liées, Données ouvertes liées, Intégration des données, Traitement de requêtes distribuées, Évaluation des requêtes fédérées, Optimisation de requêtes SPARQL.