

IONIAN UNIVERSITY

Faculty of Information Science and Informatics

Department of Archives, Library Science and Museology



Data and Knowledge Management in Cloud Computing Environment

Ph.D. Thesis

Eleftherios Kalogeros

Corfu, 2022

Supervisor

Manolis Gergatsoulis, *Professor, Ionian University*

Advisory Committee Members

Manolis Gergatsoulis, *Professor, Department of Archives, Library Science and
Museology, Ionian University*

Christos Papatheodorou, *Professor, Department of History and Philosophy of
Sciences, National and Kapodistrian University of Athens*

Timos Sellis, *Professor, Department of Computer Science and Software
Engineering, Swinburne University*

Reviewing Committee Members

Manolis Gergatsoulis, *Professor, Department of Archives, Library Science and
Museology, Ionian University*

Christos Papatheodorou, *Professor, Department of History and Philosophy of
Sciences, National and Kapodistrian University of Athens*

Spyros Sioutas, *Professor, Department of Computer Engineering & Informatics,
University of Patras*

Dimitris Fotakis, *Associate Professor, School of Electrical and Computer
Engineering, National Technical University of Athens*

Christos Nomikos, *Associate Professor, Department of Computer Science &
Engineering, University of Ioannina*

Michalis Stefanidakis, *Associate Professor, Department of Informatics, Ionian
University*

Dimitrios Tsoumakos, *Associate Professor, School of Electrical and Computer
Engineering, National Technical University of Athens*

My Ph.D. thesis represents my original research work. Where contributions of others were involved, there is every effort to indicate this clearly, with due reference to the literature, as well as acknowledgment of collaborative research and discussion.

The Ph.D. thesis was supervised by Professor Manolis Gergatsoulis, Department of Archives, Library Science and Museology, Ionian University.

Eleftherios Kalogeros

2022

To my parents for their support and patience.

Acknowledgments

The elaboration of this doctoral thesis was an interesting and fruitful experience primarily thanks to the subject it deals with, but also thanks to the people I came in contact with who were directly or indirectly involved, and who provided me with ideas, useful material and moral support.

First of all, I feel the need to express my deep gratitude to the supervisor professor of the thesis, Manolis Gergatsoulis, Professor in the Department of Archives, Library Science and Museology of the Ionian University who encouraged me to start dealing with the topic of this thesis and helped me to successfully complete it. I do not find the appropriate words to thank him for the valuable time he devoted to me, for the apt remarks during our discussions and his enthusiasm to introduce me to all aspects of our Science. His belief that in the end the whole project is possible as well as his confidence in me, encouraged me to continue my study and research and to be able today to present its results. His presence throughout my research has been a catalyst, both on an academic and on human level. I express my deepest and warmest thanks for everything I had the opportunity to learn next to him during our acquaintance and cooperation.

I also thank the members of the supervisory committee, Christos Papatheodorou, Professor in the Department of History and Philosophy of Sciences of the National and Kapodistrian University of Athens and Timos Sellis, Professor in the Department of Computer Science and Software Engineering of the Swinburne University for the knowledge they have offered me all these years as well as their trust. The contact and communication with them has been crucial in highlighting important issues of the present thesis and a source of inspiration.

I would also like to thank the co-authors of the scientific publications related to the present thesis Matthew Damigos, Assistant Professor in the Department of Archives, Library Science and Museology of the Ionian University and Christos Nomikos, Associate Professor in the Department of Computer Science & Engineering of the University of Ioannina.

Last but not least, I express my thanks to my parents and brother for their endless support and understanding.

Abstract

In this Ph.D. thesis, we study the problem of efficiently evaluating basic graph pattern (BGP) queries over a large amount of linked data (RDF data) in parallel and provide four approaches. In this context, we consider that the data graph has been partitioned into *graph segments* and the initial query Q is decomposed into a set of BGP subqueries.

In the first three approaches, the widely used MapReduce framework is used for querying a large amount of linked data.

In the first approach, a generic two-phase, MapReduce algorithm is presented. The algorithm is based on the idea that the data graph has been arbitrarily partitioned into *graph segments* which are stored in different nodes of a cluster of commodity machines. To answer a user query Q , Q is also decomposed into a set of random BGP subqueries. In the first phase, the subqueries are applied to each graph segment, in isolation, and intermediate results are computed. The intermediate results are appropriately combined in the second phase to obtain the answers of the initial query Q . The proposed algorithm computes the answers to a given query correctly, independently of a) the data graph partitioning, b) the way that graph segments are stored, c) the query decomposition, and d) the algorithm used for calculating (partial) results.

In the second approach, we present a method which focusing on the decomposition of the query Q , into a set of *generalized star queries*, which are queries that allow both subject-object and object-subject edges from a specific node, called *central node*. It is proved that each query Q can be transformed into a set of subject-object star subqueries. The data graph has also been arbitrarily partitioned into *graph segments* and a two-phase, scalable MapReduce algorithm is proposed that efficiently results from the answer of the initial query Q by computing and appropriately combining the generalized subqueries answers.

The third approach is based on the assumptions that data graphs are partitioned in the distributed file system in such a way so as replication of data triples between the data segments is allowed. Data triples are replicated in such a way so as answers of generalized star queries, can be obtained from a single data segment. One and a half phase, scalable, MapReduce algorithm is proposed that efficiently computes the answer of the initial query Q by computing and appropriately combining the subquery answers. It is proved that, under certain conditions, the query can be answered in a single MapReduce phase.

In the fourth approach, we propose an effective data model for storing RDF

data in a document database using a maximum replication factor of 2 (i.e., in the worst-case scenario, the data graph will be doubled in storage size). The proposed storage model is utilized for efficiently evaluating BGP queries in a distributed manner. Each query is decomposed into a set of *generalized star queries*. The proposed data model ensures that no joining operations over multiple datasets are required to evaluate generalized star queries. The results of the evaluation of the generalized star subqueries of a query Q are then properly combined, in order to compute the answers of the initial query Q . The proposed approach has been implemented using MongoDB and Apache Spark.

Περίληψη

Στην παρούσα διδακτορική διατριβή μελετάται το πρόβλημα της παράλληλης επεξεργασίας ερωτημάτων σε μεγάλου όγκου διασυνδεδεμένα δεδομένα (κωδικοποιημένα μέσω της RDF) μέσω τεσσάρων διαφορετικών πρωτότυπων προσεγγίσεων. Σε όλες τις προσεγγίσεις, τα αρχικά δεδομένα διασπώνται σε υπογράφους (τμήματα), το ερώτημα Q που είναι και αυτό της μορφής γράφου διασπάται σε μικρότερους υπογράφους (υποερωτήματα), και στη συνέχεια η επεξεργασία των υποερωτημάτων στα διάφορα τμήματα πραγματοποιείται παράλληλα.

Στις τρεις πρώτες προσεγγίσεις, η επεξεργασία των δεδομένων βασίζεται στο ευρέως χρησιμοποιούμενο προγραμματιστικό περιβάλλον MapReduce.

Στην πρώτη προσέγγιση, παρουσιάζεται ένας γενικός αλγόριθμος MapReduce αποτελούμενος από δύο φάσεις. Ο αλγόριθμος υλοποιείται πάνω σε μεγάλου όγκου δεδομένα που έχουν διασπαστεί και αποθηκευτεί σε διαφορετικούς κόμβους (nodes) μιας συστοιχίας υπολογιστών (cluster). Το αρχικό ερώτημα Q , διασπάται σε ένα πλήθος τυχαίων υποερωτημάτων. Στην πρώτη φάση του αλγορίθμου, το κάθε υποερώτημα επεξεργάζεται παράλληλα σε κάθε υπογράφο και τα αποτελέσματα των υποερωτημάτων συνδυάζονται κατάλληλα στη δεύτερη φάση του αλγορίθμου ώστε να υπολογιστούν τα τελικά αποτελέσματα του αρχικού ερωτήματος Q . Ο προτεινόμενος αλγόριθμος υπολογίζει τις απαντήσεις ανεξάρτητα α) από τον τρόπο διάσπασης του αρχικού γράφου, β) του τρόπου αποθήκευσης των δεδομένων, γ) του τρόπου διάσπασης του αρχικού ερωτήματος στα υποερωτήματα και δ) του αλγορίθμου που χρησιμοποιείται για τον υπολογισμό των (μερικών) αποτελεσμάτων.

Η δεύτερη προσέγγιση βασίζεται στη διάσπαση του αρχικού ερωτήματος Q σε ένα πλήθος υποερωτημάτων που αποκαλούνται ερωτήματα *γενικευμένου αστέρα*. Όλες οι ακμές αυτού του είδους ερωτημάτων έχουν έναν κοινό κόμβο είτε ως υποκείμενο, είτε ως αντικείμενο που ονομάζεται κεντρικός κόμβος. Αποδεικνύεται ότι κάθε ερώτημα Q , μπορεί να διασπαστεί σε ένα πλήθος τέτοιων υποερωτημάτων. Τα αρχικά δεδομένα και πάλι κατανέμονται στους κόμβους της συστοιχίας των υπολογιστών και ένας επεκτάσιμος αλγόριθμος MapReduce που παρουσιάζεται υπολογίζει αποδοτικά τις απαντήσεις του αρχικού ερωτήματος, αφού πρώτα υπολογίσει και στη συνέχεια συνδυάσει κατάλληλα τις απαντήσεις των υποερωτημάτων της μορφής γενικευμένου αστέρα.

Η τρίτη προσέγγιση βασίζεται στην αρχική διάσπαση του γράφου των δεδομένων με τέτοιο τρόπο ώστε να επιτρέπεται η επανάληψη ίδιων ακμών σε περισσότερα από έναν υπογράφους (τμήματα) με τέτοιο τρόπο ώστε να εξασφαλίζεται ο υπολογισμός των απαντήσεων ενός υποερωτήματος γενικευμένου αστέρα τοπικά

σε κάθε υπογράφο. Ο επεκτάσιμος αλγόριθμος MapReduce που μελετάται για τον υπολογισμό των απαντήσεων του αρχικού ερωτήματος αποτελείται από ένα και μισό στάδιο MapReduce. Παρόλα αυτά, αποδεικνύεται πως σε ειδικές περιπτώσεις τα τελικά αποτελέσματα μπορούν να απαντηθούν και σε ένα μόνο στάδιο MapReduce.

Στην τέταρτη προσέγγιση παρουσιάζεται ένα αποτελεσματικό μοντέλο οργάνωσης και αποθήκευσης των δεδομένων κατάλληλο για επεξεργασία τους σε NoSQL document βάσεις δεδομένων. Η επανάληψη ίδιων ακμών του αρχικού γράφου στους υπογράφους γίνεται και πάλι με τέτοιο τρόπο ώστε στη χειρίστη περίπτωση τα δεδομένα μετά τη διάσπασή τους να μην έχουν μέγεθος μεγαλύτερο (στη χειρίστη περίπτωση) από το διπλάσιο του μεγέθους των αρχικών δεδομένων. Το αρχικό ερώτημα διασπάται και πάλι σε ένα σύνολο από υποερωτήματα τύπου γενικευμένου αστέρα. Το προτεινόμενο μοντέλο οργάνωσης των δεδομένων εξασφαλίζει πως κάθε υποερώτημα τύπου αστέρα μπορεί να απαντηθεί μέσα σε κάθε ξεχωριστό document της βάσης δεδομένων. Οι απαντήσεις των υποερωτημάτων στη συνέχεια συνδυάζονται κατάλληλα ώστε να προκύψουν οι τελικές απαντήσεις του αρχικού ερωτήματος Q . Η προσέγγιση υλοποιήθηκε κάνοντας χρήση της NoSQL document βάσης δεδομένων MongoDB και του Apache Spark.

Contents

1	Introduction	1
1.1	Semantic Web	1
1.1.1	Resource Description Framework	3
1.1.2	SPARQL Query Language	6
1.1.3	Linked Data	8
1.2	Ph.D. Thesis Study	9
1.3	Structure of the Ph.D. Thesis	11
2	Distributed Processing Frameworks and Storage Systems for Big Data	12
2.1	Big Data	12
2.2	Apache Hadoop	13
2.2.1	Map-Reduce Programming Framework	13
2.2.2	Apache Hadoop Architecture	14
2.3	Apache Spark	14
2.3.1	Apache Spark Programming Model	15
2.4	NoSQL Databases	16
2.4.1	Key-Value Model	16
2.4.2	Document Based Model	17
2.4.3	Graph Based Model	17
2.4.4	Column Based Model	18
3	Definition of the Query Evaluation Framework	19
3.1	Data and Query Graph	19
3.2	(Total) Embeddings	21
3.3	Data and Query Graph Decomposition	21
3.4	Partial embeddings	25

3.5	Special forms of queries	27
4	Query Evaluation by Joining Partial Embeddings	29
4.1	The QEJPE Algorithm	29
4.2	A MapReduce Implementation of the QEJPE Algorithm	34
4.2.1	The Preprocessing Phase	34
4.2.2	Phase 1 of the QEJPE-algorithm	36
4.2.3	Phase 2 of the QEJPE-algorithm	39
4.3	Discussion	41
5	Query Evaluation by Decomposing Queries Into Generalized Stars	42
5.1	The eval-STARS Algorithm	42
5.2	A Map-Reduce Implementation of the eval-STARS Algorithm	45
5.2.1	Preprocessing Phase	45
5.2.2	Phase 1 of the algorithm	46
5.2.3	Phase 2 of the algorithm	52
5.3	Discussion	54
6	Query Evaluation by Data Decomposition Using Replication	56
6.1	The QE-with-Redundancy Algorithm	56
6.2	A Map-Reduce Implementation of the QE-with-Redundancy Algorithm	61
6.2.1	The preprocessing phase	62
6.2.2	Phase 1 of the algorithm	62
6.2.3	Phase 2 of the algorithm	65
6.3	Query Decomposition Algorithms	67
6.3.1	Subqueries with a limited number of variables	69
6.3.2	Reducing the number of subqueries	73
6.4	Discussion	80
7	Query Evaluation Over Linked Data Stored in a Document Database	81
7.1	Preliminaries	81
7.2	The Node-oriented Partitioning of a Data Graph	83
7.3	Decomposing Query Graphs into Generalized Star Queries	85
7.4	The Doc-based Algorithm	88
7.5	Doc-based Implementation using MongoDB and Apache Spark	89

7.5.1	Document Data Model for Storing Node-Oriented Data Graph Decomposition Segments	90
7.5.2	Efficient Parallel Partitioning Algorithm	90
7.5.3	Evaluating Graph Pattern Queries using MongoDB and Apache Spark	91
7.6	Discussion	93
8	Experimental Results	100
8.1	Evaluation of the QEJPE, eval-STARS and QE-with-Redundancy Algorithms	100
8.1.1	Scalability of the Query Evaluation Algorithms	102
8.1.2	Comparison of Query Evaluation Algorithms	104
8.1.3	Query Decomposition Algorithms Evaluation	107
8.2	Evaluation of the Doc-based Algorithm	111
9	Related work	117
9.1	Centralized Environments for Querying Linked Data	117
9.2	Using Distributed Storage and Processing	118
9.2.1	Query Answering Using MapReduce	120
9.2.2	Query Answering Using Spark	123
9.2.3	Using NoSQL Databases	124
9.2.4	Other Distributed Query Evaluation Engines	125
10	Conclusions	128

List of Tables

8.1	Description of the Datasets	101
8.2	Description of the queries evaluated over datasets of different sizes	102
8.3	Query evaluation in datasets of different sizes (in seconds)	104
8.4	Query evaluation in D4 dataset for different compute nodes size (seconds)	104
8.5	Description of Watdiv queries	106
8.6	Execution time per query	106
8.7	Average execution time per query type (seconds)	106
8.8	Evaluation using QE-with-Redundancy and different query de- compositions (Seconds)	110
8.9	Datasets characteristics	112
8.10	Execution times for 17 WatDiv benchmark queries (seconds). . .	114
8.11	Average execution times of the queries grouped according to the benchmark query categorization (seconds)	114

List of Figures

1.1	Semantic Web Stack Technologies	2
1.2	A RDF graph example.	4
1.3	A SPARQL BGP Query Example.	7
2.1	MapReduce Programming Model	13
3.1	(a) A data graph, and (b) a query graph	21
3.2	An embedding of the query graph Q in the data graph G	22
3.3	3-triple partition of the data graph G of Fig. 3.1(a).	24
3.4	Query decomposition.	25
5.1	Query decomposition into star queries.	44
6.1	An s-decomposition of the data graph G of Fig. 3.1.	58
6.2	Min-res Query decomposition.	73
6.3	Min-subquery decomposition.	74
6.4	Min-subquery vs. Max-degree decomposition	76
7.1	Examples of generalized star queries.	83
7.2	Two node-partition segments G_1 , G_2 obtained from the graph G of Fig. 3.2.	85
7.3	An algorithm for decomposing a query graph into generalized star queries.	95
7.4	(a) Document data model, (b) Segment represented by a JSON document.	96
7.5	A node-partition algorithm.	97
7.6	An algorithm for transforming generalized star query to MongoDB query.	98

7.7	The DAG operations for the full query processing. An example with a query decomposed into three generalized star subqueries. .	99
8.1	Query evaluation in terms of the size of dataset (in seconds) . . .	103
8.2	Query evaluation in terms of compute nodes size per algorithm (seconds)	105
8.3	Comparison of query evaluation algorithms for a variety of query types (seconds)	108
8.4	Queries used to compare decomposition algorithms	109
8.5	Query decomposition algorithms evaluation (seconds)	110
8.6	Query evaluation experimental results	115
8.7	Query type summarization evaluation experimental results. . . .	115
8.8	Scaling analysis.	116

Chapter 1

Introduction

In this Chapter we present an introduction about the Semantic Web innovation and technologies as well the Ph.D. thesis study.

1.1 Semantic Web

The World Wide Web (WWW) constitutes a global information system that is used to share data of various formats made up of interlinked documents and files. The interlinked documents and files in WWW are identified by Uniform Resource Locators (URL) that provide the location and the way (protocol) from which a document can be retrieved. The existence of hyperlinks between the documents offers to users the ability to navigate between them. The simple architecture of WWW turned it to a rapidly growing information system where users from all over the world use it daily to meet their information needs.

The initial architecture of the WWW based on linked documents (Web of Documents), because of its simplicity, offered the opportunity for rapidly growth, nevertheless the lack of semantic organization of data, produce difficulties in seeking information. The extension of the current web to the Semantic Web (SW) aims to convert the WWW from a global place of interlinked documents to a global place of interlinked data (Web of Data). In SW, data resources and their semantic links are identifying using Internationalized Resource Identifiers (IRI) [85], a generalization of URIs, and data resources are semantically organized in such a way that information about resources can be processed by machines. The encoding of semantics of the resources in SW takes place using data models, query

languages, vocabularies, ontologies and rules. Resource Description Framework (RDF) [84], SPARQL query language [115, 102], Web Ontology Language (OWL) [86] are standardized SW technologies from World Wide Web Consortium (W3C).

Figure Fig. 1.1 [50] depicts the SW stack technologies.

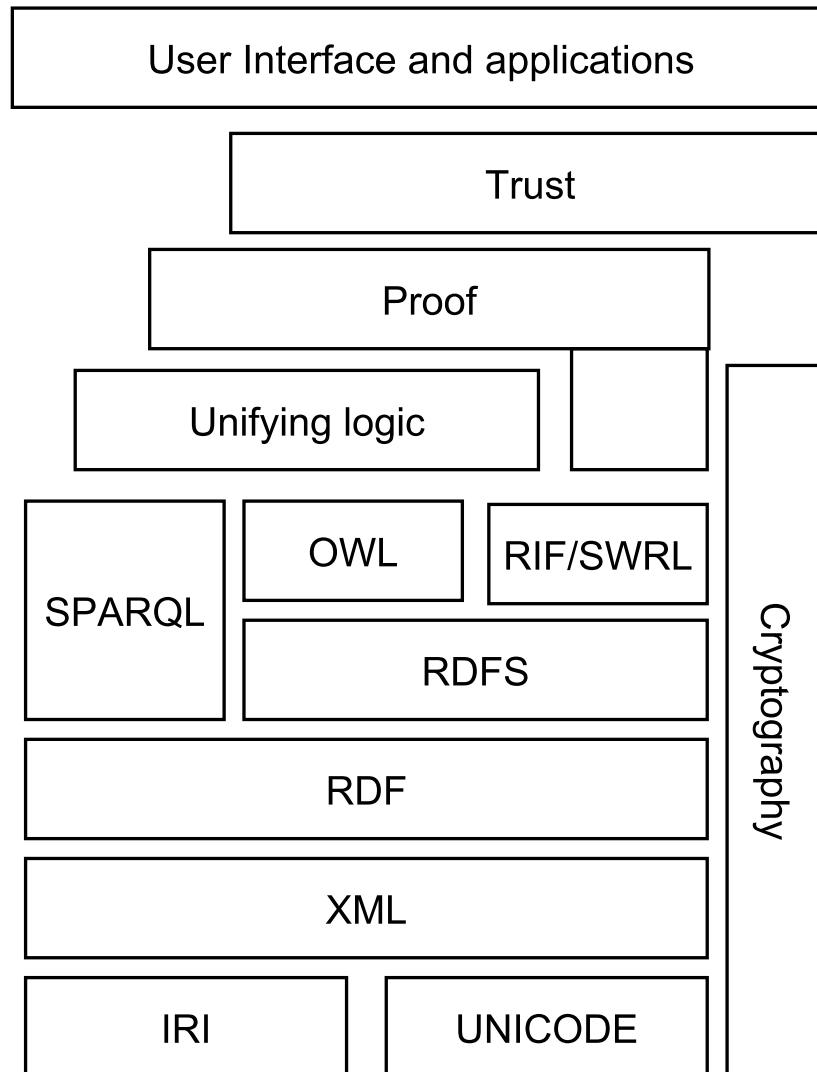


Figure 1.1: Semantic Web Stack Technologies

Resource Description Framework (RDF) is the data model used in SW to describe resources in the form of a graph. RDF Schema (RDFS) [106] is a semantic extension of RDF that provides a vocabulary for RDF to organize the resources into groups (classes) and to describe the relations between resources (properties). Web Ontology Language (OWL) extends RDFS for describing properties and classes in order to extend the limited expressiveness of RDF Schema. RDFS and OWL are based on description logic and enhance the SW with reasoning. SPARQL is the SW query language, it is used to query any RDF data even those that include RDFS and OWL statements. Querying language is necessary to retrieve information for semantic web applications. Rule Interchange Format (RIF) is a rule interchange format for exchanging rules among rule systems.

1.1.1 Resource Description Framework

The Resource Description Framework (RDF) is a graph data model (a W3C recommendation since 1999) for describing abstract or physical resources in the world wide web (WWW). RDF provides the data model for the Semantic Web. RDF data model consists of simple statements called RDF data triples by using three components called subject, predicate (or property) and object respectively. The value of a subject in a triple is a URI reference (identifier) of the resource, the predicate is a URI reference that represents the relation between the subject and object and the object is a URI reference of a resource or Literal. Literals may be plain literals (e.x. “Semantic Web”) or typed literals (e.x. “Semantic Web”^{xs}:string). An RDF triple’s subject can be the object of another triple and this results in many data triples to form a directed graph. The data graph nodes are the subject and object values and the arcs are the values of the predicates that connect the subject and the object. When a node either subject or object of a data triple is no IRI or literal is called a blank node. Blank nodes are used to imply the existence of a resource, without using a URI to identify the resource.

Formally, an RDF triple and RDF graph (*RDF dataset* or simple a *dataset*) can be defined as follows:

Definition 1.1.1. *Let I be disjoint infinite set of URI references, B be disjoint infinite set of blank nodes and L be an infinite set of literals. A RDF triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called a RDF triple. A RDF graph is a labeled, directed graph defined as a non-empty set of RDF triples.*

A RDF graph G' is a *RDF subgraph* of a RDF graph G if $G' \subseteq G$. It is easy to separate a RDF graph into smaller subgraphs separating its RDF triples. Respectively the union of RDF subgraphs into one RDF graph is achieved by merging all RDF triples of all subgraphs.

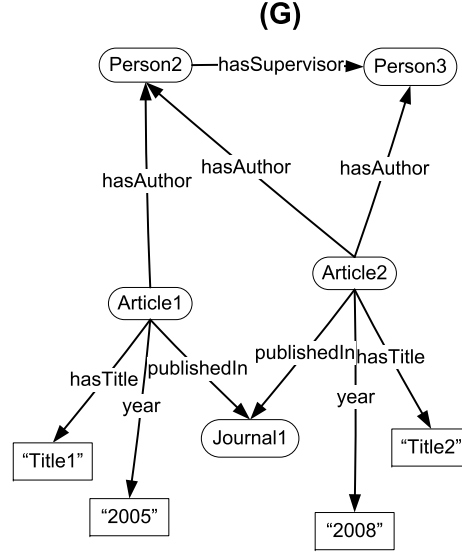


Figure 1.2: A RDF graph example.

Figure Fig. 1.2 illustrates an example of RDF graph consists of ten RDF triples. A node (subject or object), which is URI or blank node, is represented as a rounded rectangle. An object which is a literal is represented by a conventional rectangle. A triple (s, p, o) is represented by the directed edge $s \xrightarrow{p} o$, labeled by p , from the node representing the subject s to the node representing the object o . The edge is also referred to as an *incoming* edge of o , as well as an *outgoing* edge of s .

Strings with initial lowercase letters represent predicates while strings with initial uppercase letters denote URIs labeling graph nodes (i.e. subjects or objects). The literals are represented as strings enclosed in double quotes.

Various serialization formats exist for RDF data graphs. RDF/XML [53], N-Triples [39], Turtle [38], TriG [114] and N-Quads [37], JSON-LD [46] and RDFa [118] are the most common.

RDF/XML is defined by W3C to serialize RDF data graphs using XML. N-Triples uses three values in the form $\langle \text{subject} \rangle \langle \text{predicate} \rangle \langle \text{object} \rangle$. to express a RDF triple. Below is expressed in RDF/XML and N-Triples

serialization formats the RDF graph depicted in figure Fig. 1.2.

Example 1.1.1. *RDF/XML serialization example*

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:ex="https://example.org/">
  <rdf:Description rdf:about="https://example.org/Article1">
    <ex:hasTitle>Title1</ex:hasTitle>
  </rdf:Description>
  <rdf:Description rdf:about="https://example.org/Article1">
    <ex:year>2005</ex:year>
  <rdf:Description rdf:about="https://example.org/Article1">
    <ex:hasAuthor rdf:resource="https://example.org/Person2"/>
  </rdf:Description>
  <rdf:Description rdf:about="https://example.org/Article2">
    <ex:hasAuthor rdf:resource="https://example.org/Person2"/>
  </rdf:Description>
  <rdf:Description rdf:about="https://example.org/Article2">
    <ex:hasAuthor rdf:resource="https://example.org/Person3"/>
  </rdf:Description>
  <rdf:Description rdf:about="https://example.org/Person4">
    <ex:hasSupervisor rdf:resource="https://example.org/Person1"/>
  </rdf:Description>
  <rdf:Description rdf:about="https://example.org/Person2">
    <ex:hasSupervisor rdf:resource="https://example.org/Person3"/>
  </rdf:Description>
  <rdf:Description rdf:about="https://example.org/Article1">
    <ex:publishedIn rdf:resource="https://example.org/Journal1"/>
  </rdf:Description>
  <rdf:Description rdf:about="https://example.org/Article2">
    <ex:publishedIn rdf:resource="https://example.org/Journal1"/>
  </rdf:Description>
  <rdf:Description rdf:about="https://example.org/Article2">
    <ex:year>2008</ex:year>
  </rdf:Description>
  <rdf:Description rdf:about="https://example.org/Article2">
    <ex:hasTitle>Title2</ex:hasTitle>
```

```
</rdf:Description>
</rdf:RDF>
```

Example 1.1.2. *N-Triple serialization example*

```
<https://example.org/Article1> <https://example.org/hasTitle>
"Title1" .
<https://example.org/Article1> <https://example.org/year>
"2005" .
<https://example.org/Article1> <https://example.org/hasAuthor>
<https://example.org/Person2> .
<https://example.org/Article2> <https://example.org/hasAuthor>
<https://example.org/Person2> .
<https://example.org/Article2> <https://example.org/hasAuthor>
<https://example.org/Person3> .
<https://example.org/Person2> <https://example.org/hasSupervisor>
<https://example.org/Person3> .
<https://example.org/Article1> <https://example.org/publishedIn>
<https://example.org/Journal1> .
<https://example.org/Article2> <https://example.org/publishedIn>
<https://example.org/Journal1> .
<https://example.org/Article2> <https://example.org/year>
"2008" .
<https://example.org/Article2> <https://example.org/hasTitle>
"Title2" .
```

1.1.2 SPARQL Query Language

SPARQL is the W3C recommendation query language for RDF. It is based on graph pattern matching. Besides the basic graph pattern matching (BGP) parts, it includes optional graph pattern matching parts, union of pattern matching parts, nesting pattern matching parts, and filtering. BGP queries consist of one or more query triple patterns, where all the query triple patterns must match in the RDF graph.

A pattern matching query is used to find all subgraphs of a data graph that are isomorphic (i.e., equivalent in both structure and labels) to a given pattern graph

SPARQL has four query types. These query types are:

1. SELECT query type that returns the selected variables that are bounded in the query pattern.
2. CONSTRUCT query type that returns the RDF subgraph constructed by substituting variables in a set of triple templates.
3. ASK query type that returns a boolean related to if the query pattern has at least one result or not.
4. DESCRIBE query type that returns an RDF graph that describes the resources.

Formally, a query triple pattern and a BGP query graph pattern can be defined as follows:

Definition 1.1.2. Let V be disjoint infinite set of variables, I be disjoint infinite set of URI references, B be disjoint infinite set of blank nodes and L be an infinite set of literals. A query triple pattern $(s, p, o) \in (V \cup I \cup B) \times I \times (V \cup I \cup B \cup L)$ is called a query triple pattern. A BGP query graph pattern is a set of query triple patterns that must all match in the RDF graph.

In figure Fig. 3.2 is depicted a SPARQL BGP query example. The query consists of eight query triples. The query consists of five variables ($?P1, ?J, ?P2, ?A, ?T$). All the other nodes and the predicates are constants either URI values or Literals. One answer of the query is:

$\langle (?P1, ?J, ?P2, ?A, ?T) \rangle = \langle (Person2, Journal1, Person3, Article2, "Title1") \rangle$.

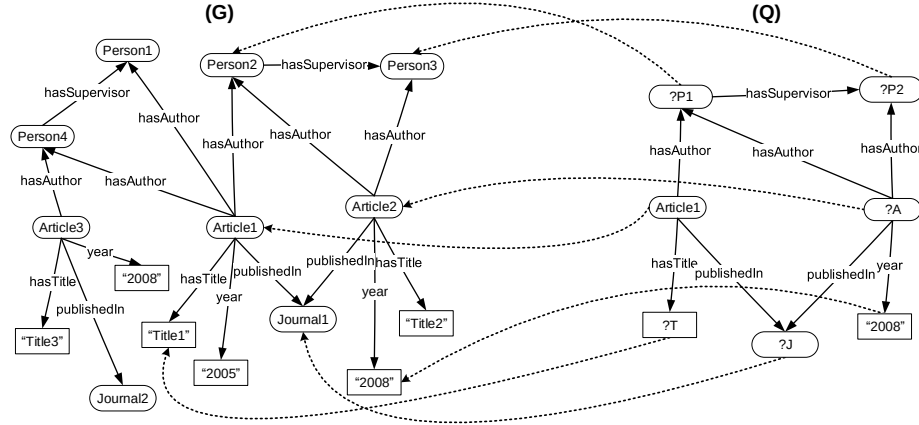


Figure 1.3: A SPARQL BGP Query Example.

Example 1.1.3. *The SPARQL syntax of the BGP query depicted in figure Fig. 3.2 is:*

```
SELECT ?P1 ?J ?P2 ?A ?T
WHERE {
  ?P1 <hasSupervisor> ?P2 .
  <Article1> <hasAuthor> ?P1 .
  <Article1> <publishedIn> ?J .
  <Article1> <hasTitle> ?T .
  ?A <hasAuthor> ?P1 .
  ?A <hasAuthor> ?P2 .
  ?A <publishedIn> ?J .
  ?A <year> "2008" .
}
```

1.1.3 Linked Data

Linked Data (LD) [35] refers to a set of best practices invented by Tim Berners-Lee for publishing data on the Web. These principles are:

1. Use URIs to identify resources
2. Use HTTP URIs to provide the opportunity those resources to be searchable
3. Provide useful information, using the standards like RDF, RDFS, OWL, SPARQL for each resource
4. In each dataset include semantic links (RDF links) to other datasets so that user can discover related resources

The last decade, LD became a widespread method for publishing interlinked data based on RDF. Besides the LD that produced for private purposes, there is a daily production of LD datasets which are released under an open license and they are mentioned as Linked Open Data (LOD). The datasets contain data from various domains. As mentioned on the web page <https://lod-cloud.net>, LOD contained as of May 2020, 1,301 datasets with 16,283 links between them with these values increasing on a daily basis. Some of the larger LOD datasets are: DBpedia¹, which data are derived from the structured information of Wikipedia and turns it into a rich knowledge base that can be used to answer

¹<http://dbpedia.org/>

expressive queries consists of 9,500,000,000 triples, Wikidata² with 96,942,861 triples, GeoNames³ with 93,896,732 triples and YAGO⁴ with 120,000,000 triples. LOD datasets can be downloaded as dump RDF files and also provide a public SPARQL Endpoint from which users can search for information using the SPARQL query language.

For example, DBpedia offers the DBpedia Ontology, a cross-domain ontology, which has been manually created based on the infoboxes within Wikipedia web pages to organize resources and the public DBpedia SPARQL Endpoint⁵ for querying. Despite the weaknesses in the way it is organized, DBpedia is one of the most important knowledge datasets on the web.

1.2 Ph.D. Thesis Study

Nowadays, the amount of information published through the WWW is rapidly growing; revealing a challenge for efficient analysis of a large amount of linked data. Furthermore, semantic annotations are usually used when exchanging and publishing data; where the RDF data model and LD paradigm are widely used to structure the data along with its semantics, in a unified manner.

Storing and analyzing RDF data has been extensively investigated in the past (e.g., [108, 74, 95, 41]), but it still remains a significant challenge, especially, due to the challenge of analyzing big data.

Centralized RDF data stores (e.g. [9, 23]), which are widely used to store and manage RDF and LD, usually utilize relational structures for persisting data. However, most of the single-machine systems used to store and manage RDF data (e.g., [90, 133]) lack scalability and efficiency in processing and analyzing large amounts of RDF data graphs [26]. To overcome the scalability and efficiency issues, in case of managing a large amount of RDF data, several approaches have been proposed; either using distributed NoSQL databases (e.g., [87, 104, 78]), or using distributed computation frameworks, such as MapReduce and Spark, over the RDF data stored in a distributed file system (e.g., [111, 49, 107, 31, 113, 44]).

This Ph.D. thesis proposes and evaluates four different approaches for efficiently evaluating basic graph pattern (BGP) queries over a large amount of linked data (RDF data) in parallel. We present the approaches in Chapters

²<https://www.wikidata.org/>

³<http://www.geonames.org/>

⁴<http://yago-knowledge.org>

⁵<https://dbpedia.org/sparql>

4,5,6 and 7. In all approaches, the data graph has been partitioned into *graph segments* and the initial query Q is decomposed into a set of BGP subqueries. The processing of the subqueries over the graph segments and the combination of the intermediate results of them are performed in parallel. The distributed processing frameworks and storage systems for big data which are used in the four query evaluation approaches are described in Chapter 2.

In the first approach, described in Chapter 4, a generic two-phase, MapReduce algorithm is presented. In the first phase, the subqueries are applied to each graph segment, in isolation, and intermediate results are computed. The intermediate results are appropriately combined in the second phase to obtain the answers of the initial query Q . The proposed algorithm computes the answers to a given query correctly, independently of a) the data graph partitioning, b) the way that graph segments are stored, c) the query decomposition (random BGP subqueries), and d) the algorithm used for calculating (partial) results of each subquery. Preliminary results of this approach were presented in: [56].

In the second approach, described in Chapter 5 we present a method which focused on the decomposition of the query Q , into a set of *generalized star queries*, which are queries that allow both subject-object and object-subject edges from a specific node, called *central node*. It is proved that each query Q can be transformed into a set of subject-object star subqueries. A two-phase, scalable MapReduce algorithm is proposed that efficiently results the answer of the initial query Q by computing and appropriately combining the generalized subqueries answers. The algorithm also computes the answers to a given query correctly, independently of a) the data graph partitioning, b) the way that graph segments are stored, c) the query decomposition (different generalized subqueries), and d) the algorithm used for calculating (partial) results of each subquery. Preliminary results of this approach were presented in: [93].

The third approach, described in Chapter 6, is based on the assumptions that data graphs are partitioned in the distributed file system in such a way so as replication of data triples between the data segments is allowed. Data triples are replicated in such a way so as answers of a subset of generalized subqueries called subject-object star queries, can be obtained from a single data segment. One and a half phase, scalable, MapReduce algorithm is proposed that efficiently computes the answer of the initial query Q by computing and appropriately combining the subquery answers. It is proved that, under certain conditions, the query can be answered in a single MapReduce phase. Preliminary results of this approach were presented in: [71].

In the fourth approach, described in Chapter 7, we propose an effective data model for storing RDF data in a document database using a maximum replication factor of 2 (i.e., in the worst-case scenario, the data graph will be doubled in storage size). The proposed storage model is utilized for efficiently evaluating BGP queries in a distributed manner. Each query is decomposed into a set of *generalized star queries*. The proposed data model ensures that no joining operations over multiple datasets are required to evaluate generalized star queries. The results of the evaluation of the generalized star subqueries of a query Q are then properly combined, in order to compute the answers of the initial query Q . The proposed approach has been implemented using MongoDB and Apache Spark. Preliminary results of this approach were presented in: [72, 73].

1.3 Structure of the Ph.D. Thesis

The Ph.D. thesis is organized in the following Chapters.

Chapter 1 and Chapter 2 are introductory Chapters and they are focused on the linked data and big data processing. More specifically, in chapter 1 we present an introduction to SW innovation and SW web technologies (RDF, SPARQL, Linked Data) while in Chapter 2 we present the distributed processing frameworks and storage systems for big data, Apache Hadoop, Apache Spark and NoSQL databases. Storing and querying big volumes of linked data is the aim of this Ph.D. thesis. Chapter 3 contains the preliminaries about data and query graphs while in each of the Chapters 4, 5, 6 and 7 we present the different approaches studied in this Ph.D. thesis. Detailed experimental results for all approaches are analyzed in Chapter 8. In Chapter 9, we present the related work about linked data parallel processing. Chapter 10 contains the conclusions of the Ph.D. thesis.

Chapter 2

Distributed Processing Frameworks and Storage Systems for Big Data

In this Chapter, we present two frameworks for scalable and parallel processing big volumes of data, Apache Hadoop [5] and Apache Spark [11], and the most usual non-relational data models for storing and querying big data. The programming framework of Apache Hadoop, MapReduce [48] was used for the implementation of the query evaluation algorithms presented in Chapter 4, Chapter 5 and Chapter 6. Apache Spark and the NoSQL document database MongoDB were used for the implementation of the query evaluation algorithm presented in Chapter 7.

2.1 Big Data

Big data [80] is high-volume, high-velocity and/or high-variety data. Big data structures may vary from relational data structures to semi-structured or unstructured data. According to big data characteristics, big data is impossible to be processed in a single computing system as vertical scaling has limitations. Big data processing can utilize horizontal scaling for better scalability. In horizontal scaling, data is partitioned into a cluster of commodity machines, which require storage and processing methods that are optimized for parallel processing. There-

fore, in the last decades, parallel processing systems have been developed for big data processing as well as databases that do not necessarily use the relational data model.

In the next Sections of Chapter 2 are mentioned two parallel processing frameworks Apache Hadoop and Apache Spark and the most usual non-relational data models for storing and querying big data.

2.2 Apache Hadoop

2.2.1 Map-Reduce Programming Framework

MapReduce [48] is a programming framework, built by Google, for processing large datasets in a distributed manner. Creating a MapReduce job, the user defines two functions, *map* and *reduce*, which run in each cluster node, in isolation. The map function is applied on one or more files, in the distributed file system (DFS), and results $\langle \text{key}, \text{value} \rangle$ pairs. This process is called *map task*. The nodes that run the Map tasks are called *mappers*. The *master controller* directs the pairs emitted by mappers to the *reducers* (i.e., the nodes that apply the reduce function on the pairs) such that all pairs with the same key initialize a single reduce process, called *reduce task*. The reduce tasks apply the reduce function on the input pairs and also result in $\langle \text{key}, \text{value} \rangle$ pairs. This procedure describes a *MapReduce phase*. Furthermore, the output of the reducer can be set as the input of a map function, which gives to the user the flexibility to create procedures of multiple steps.

Figure Fig. 2.1 depicts the MapReduce programming model.

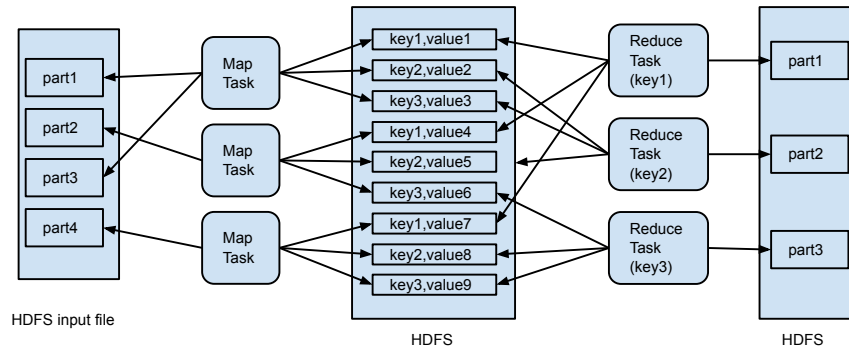


Figure 2.1: MapReduce Programming Model

2.2.2 Apache Hadoop Architecture

Apache Hadoop (Hadoop) [5] is an open-source software framework for scalable and parallel processing big volumes of data on a cluster of commodity servers using the programming model MapReduce. Hadoop architecture is divided into three layers, the Hadoop Distributed File System (HDFS), the Apache YARN (YARN) [6] and the MapReduce engine.

HDFS is the distributed file system of Hadoop. Data files imported in HDFS are separated into smaller units called blocks (default block size is 128MB or 256MB) and stored in a distributed manner. The files in HDFS are managed in a master-slave architecture. In a typical Hadoop cluster, there are two different types of nodes, NameNodes and DataNodes. NameNode keeps the location of the blocks of the files that are stored in the DataNodes. HDFS is a fault tolerance file system as each block is replicated and stored on different DataNodes. The number of copies (replication factor) of each block can be configured to any value. In a typical installation, the replication factor is set to three.

MapReduce engine is the data processing layer of Hadoop. Users develop MapReduce programs (Jobs), using the MapReduce programming model, in various programming languages like Java, Ruby and Python. The input file of the MapReduce job is usually a set of files stored on HDFS or data stored in various databases.

Hadoop runs the map and the reduce functions of a job by dividing them into tasks, the map tasks and reduce tasks running all in parallel. YARN (Yet Another Resource Negotiator) is the resource management system of Hadoop. YARN was introduced in version two of Hadoop as a task scheduler and run on nodes in the cluster. If a task fails, it will be automatically rescheduled to run on a different node providing fault tolerance to the system.

2.3 Apache Spark

Apache Spark (Spark) [11, 130] is also a parallel and fault tolerant programming framework focusing on efficiently processing large amount of data in distributed manner. Spark extends MapReduce framework in the following main aspects:

- It utilizes in-memory computations more efficiently and reduces communication with physical storage. In particular, MapReduce writes the data to the physical storage after each operation, while Spark aims to copy

the data from the physical storage into memory and performs most of its operations in memory.

- Unlike MapReduce, where the processing is achieved through a sequence of MapReduce steps, Spark supports directed acyclic graph (DAG) of operations (enabling flexible flows of operations).
- Spark supports lazy evaluation (i.e., the execution is performed once an action step is submitted), which provides a different way of achieving fault tolerance, as well as enables better optimization of execution plan.

Spark allows programs input to be files stored in HDFS or data stored in various databases. Due to the aforementioned features, Spark could be one hundred time faster than Hadoop.

2.3.1 Apache Spark Programming Model

Spark relies parallel computation and fault tolerance on the Resilient Distributed Dataset (RDD) [129], which represents a distributed collection of elements partitioned across the nodes of the cluster.

Spark also provides a high level programming paradigm in multiple programming languages and categorizes the available operations in two main categories: transformations and actions. Transformations represent the type of operations which transform an RDD to another RDD. Actions, on the other hand, represent the operators triggering execution and producing a resulting dataset. Transformations in SPARK are lazy, that is they do not perform any processing tasks. Transformations are computed when an action requires a result to be returned. This design enables Spark to run more efficiently. Spark also supports working with key-value pairs, similar to MapReduce, as well as it distinguishes operations that are typically applied over each element of an RDD, on the elements of each cluster node, and operations that require re-shuffling of the data.

It is worth mentioning that Spark also provides an table-like abstraction, called DataFrame, as an additional immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. DataFrames also support optimized operations like select columns, filtering and joins over the distributed data that make large datasets processing much easier, and allow Spark users to impose a structure onto a distributed collection of data.

Additionally, Spark includes the Spark SQL module for processing structured data and the GraphX module that extends the Spark RDD for graph processing. Since in this Ph.D. thesis we do not use all RDD transformations and actions as well Spark SQL and GraphX, detailed information from previous could be found in the official documentation of Spark.

2.4 NoSQL Databases

NoSQL (Not only SQL) databases [47] mainly refer to the database management systems that do not conform to the conventional relational data model. NoSQL databases are classified in the following four main categories, according to the data model they use: key-value databases, wide-column databases, document databases, and graph databases. The main feature of the NoSQL databases is that they support data partitioning methods into a cluster of machines (cluster nodes). The most usually used method of partition is the horizontal partitioning, where data is separated into partitions (or shards). Vertical partitioning divides data based on predefined groups of columns that are accessed together (e.g., column-families in wide-column stores) into partitions. Data partitioning methods offers scalability to the system and better query evaluation performance. Because of the distributed capabilities of the NoSQL databases, these systems respond the CAP Theorem [57] as they can only guarantee two of the following three properties, Consistency, Availability and Partition Tolerance. Most NoSQL databases support connectors (i.e., libraries) which allow those databases to be used as either an input source or output destination, for both Hadoop MapReduce and Spark jobs.

2.4.1 Key-Value Model

Key-Value databases (or Key-Values stores) can be considered as the most simple NoSQL databases. The data model they use is key, value pairs. In the key-value pair data model, the value represents data and the key is a unique identifier of the value. According to the simplicity of the data model, Key-value stores offer high scalability and efficient key-based search algorithms. Most popular key-value stores are the in-memory database Memcached [16] that keep data in memory providing fast searching methods, the persistent key-value database Riak [21], and the hybrid databases, such as Redis [20] that use both in memory (cache) and persistent storage methods. The value in most key-value stores is

not limited to strings but can be a number of data structures including hashes, lists, sets, etc.

2.4.2 Document Based Model

Document databases (or Document Stores) are document-oriented storage and query systems. Documents are organized into collections. Each document has semi-structured data either in JSON [15] or XML [96] encoding format. Among the most popular databases in this category are MongoDB [17, 43] and Couchbase [13]. Document databases are usually schema-less systems as it is not required, documents to have the same schema and the structure of the documents could vary from document to document and can be changed over time. Additionally, they support indices and offer balanced horizontal scaling as each document size is typically small. The basic disadvantage is the lack of supporting efficient joins over the stored collections. MongoDB stores and manages the documents data as binary JSON-like documents (BSON documents). The ability of Document Stores and particularly MongoDB for storing and querying linked data is studied in Chapter 7. A collection in a MongoDB database is a set of documents. Each document can have many key-value pairs. The value can also be a key-value pair so data are organized into a hierarchical manner into each document. MongoDB supports horizontal scaling and powerful indexing strategies. The architecture of a MongoDB shared cluster consists of (i) the servers called shards, each one contains a subset of the documents, (ii) the servers called *mongos* which route the query to the cluster and the (iii) the configuration servers that store metadata and configuration settings for the cluster. MongoDB cluster is fault tolerant as replication can be used to all servers. MongoDB supports a JSON like a query language and indices to one or more keys of the documents for better query performance. MongoDB support plugin [14] which allow being used as either an input source or output destination, for both Hadoop MapReduce and Spark jobs. MongoDB guarantee consistency and partition tolerance (CP)

2.4.3 Graph Based Model

Graph databases (or Graph Stores) are graph-oriented storage, query and analytics systems. These systems efficient organize data as graphs consist of vertices representing entities and edges representing relationships between them and they are used for efficient entity relationship traversals. There are two main categories of graph databases, the *property graph databases* in which data, in the

key-value pair model, can be stored in every edge and relationship (e.g. Neo4J [18] and Apache TinkerPop [12]) and the *knowledge databases* which use semantic graphs using the RDF data model called RDF databases or simple triple stores (e.g. Virtuoso, RDF-3X). Graph databases use graph-pattern matching query languages. SPARQL is the only officially standardized query language. Cypher and Gremlin are property graphs query languages designed for Neo4J and the Apache TinkerPop respectively.

2.4.4 Column Based Model

Wide Column databases (or Wide Column Stores) are designed for storing data tables as sections of columns of data, rather than as rows of data. Wide Column Stores offer very high performance and a highly scalable architecture. Among the most popular databases in this category are Cassandra [3, 79] and HBase [7, 55]. In this Ph.D. thesis, storing and querying linked data using Wide Column Stores was not studied.

Chapter 3

Definition of the Query Evaluation Framework

In this chapter we (formally) define the main notions that we will use in the following Chapters of this Ph.D. thesis.

3.1 Data and Query Graph

Let U_{so} and U_p be two countably infinite disjoint sets of URI references, L be a countably infinite set of (plain) literals¹ and V be a countably infinite set of variables. In the following, we define two types of graphs, *data graphs* and *query graphs*. The former describes the data model and the latter determines the form of the query expressions over the stored data.

Definition 3.1.1. A triple $(s, p, o) \in U_{so} \times U_p \times (U_{so} \cup L)$ is called a *data triple*. In a data triple $t = (s, p, o)$, s is called the *subject*, p the *predicate* and o the *object* of t . A *data graph* G is a non-empty set of data triples. A data graph G' is a subgraph of a data graph G if $G' \subseteq G$.

Definition 3.1.2. A triple $(s, p, o) \in (U_{so} \cup V) \times U_p \times (U_{so} \cup L \cup V)$ is called a *query triple*. In a query triple $q = (s, p, o)$, s is called the *subject*, p the *predicate* and o the *object* of q . A *query graph* (or simply a *query*) Q is a nonempty set of query triples. The output pattern $O(Q)$ of a query graph Q is the tuple $(X_1, \dots, X_n)$, with $n \geq 0$, of all the variables appearing in Q . A query

¹In this Ph.D. thesis we do not consider typed literals

Q is said to be a Boolean query if $n = 0$. A query graph Q' is a subquery of a query graph Q if $Q' \subseteq Q$.

Definition 3.1.3. Let G be a data or query graph. A directed path (or simply path) in G is a sequence of triples $(v_0, p_1, v_1), (v_1, p_2, v_2), \dots, (v_{n-1}, p_n, v_n)$ in G , where $n \geq 1$. The length of the path is the number n of triples in the path. A finite directed path always has a start node which corresponds to the subject of its first triple, and an end node which is the object of its last triple. A cycle is a path in which the start node and the end node are the same. A path with no repeated nodes (i.e. without cycles) is called a simple path.

The set of nodes of a data graph G (resp. a query graph Q), denoted by $\mathcal{N}(G)$ (resp. $\mathcal{N}(Q)$), is the set of elements of $U_{so} \cup L$ (resp. $U_{so} \cup L \cup V$) that occur in the triples of G (resp. Q). The set of edge labels of a data graph G (resp. a query graph Q), denoted by $\mathcal{E}(G)$ (resp. $\mathcal{E}(Q)$), is the set of elements of U_p that occur in the triples of G (resp. Q). Finally, the set of variables in a query Q is denoted by $\mathcal{V}(Q)$.

Notice that the data graphs defined above correspond to ground RDF graphs defined in [61]. Notice also that query graphs correspond to Basic Graph Patterns (BGP) SPARQL queries [115]. In the first three approaches of this Ph.D. thesis, presented in Chapter 4, Chapter 5 and Chapter 6 we do not allow queries with variables in the place of predicates. However, the three query evaluation algorithms proposed in this Chapters of the Ph.D. thesis can be easily extended to allow such variables.

Data and query graphs are graphically represented as follows: A node (subject or object), which is a URI or a variable, is represented as a rounded rectangle, while an object which is a literal is represented by a rectangle. Each triple (s, p, o) is represented by a labeled edge $s \xrightarrow{p} o$ connecting the nodes s and o .

In this Ph.D. thesis, we use strings with initial lowercase letters to represent elements in U_p (i.e., URIs corresponding to predicates), while strings with initial uppercase letters denote elements in U_{so} (i.e., URIs corresponding to objects and subjects). Literals are represented as strings enclosed in double quotes. Finally, we assume that variables are represented by strings whose first symbol is the question mark symbol (?).

Example 3.1.1. Fig. 3.1(a) depicts a data graph showing information about three journal papers, their authors and the relationships between the authors. Fig. 3.1(b) shows a query graph.

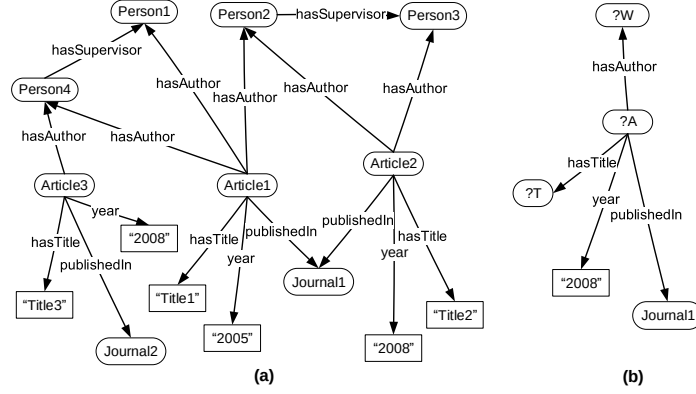


Figure 3.1: (a) A data graph, and (b) a query graph

3.2 (Total) Embeddings

To show how the queries are evaluated over a data graph we use the notion of *total embeddings* defined as follows.

Definition 3.2.1. A (total) embedding of a query graph Q in a data graph G is a total mapping $e : \mathcal{N}(Q) \rightarrow \mathcal{N}(G)$ with the following properties:

1. For each node $v \in \mathcal{N}(Q)$, if v is not a variable then $e(v) = v$.
2. For each triple $(v_1, p, v_2) \in Q$, the triple $(e(v_1), p, e(v_2))$ is in G .

The tuple $(e(X_1), \dots, e(X_n))$, where $(X_1, \dots, X_n)$ is the output pattern of Q , is said to be an answer to the query Q .

Example 3.2.1. Fig. 3.2 depicts an embedding of the query graph Q in data graph G , where Q and G are the graphs appearing in Fig. 3.1. The answer obtained by this embedding is $(?A, ?W, ?T) = (\text{Article2}, \text{Person3}, \text{"Title2"})$. Notice that a second embedding exists giving the answer $(?A, ?W, ?T) = (\text{Article2}, \text{Person2}, \text{"Title2"})$.

3.3 Data and Query Graph Decomposition

A crucial problem, when we use a cluster of computer nodes to evaluate queries, is how to distribute the data in the computers of the cluster as well as how

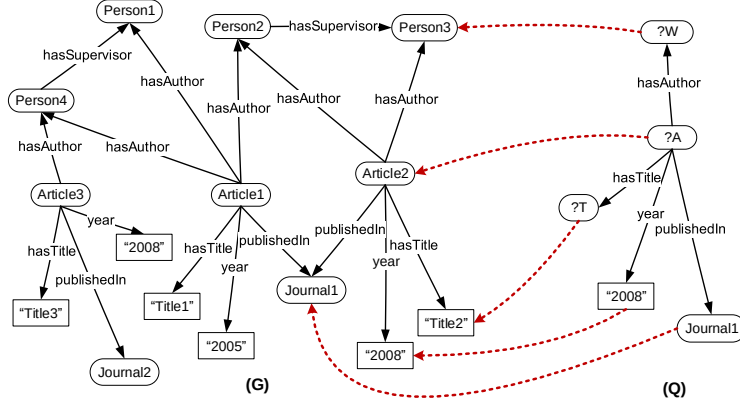


Figure 3.2: An embedding of the query graph Q in the data graph G .

to compute the queries on the distributed data. In this Section we define the concept of *data* and *query graph decomposition*.

Definition 3.3.1. A data (resp. query) graph decomposition of a data (resp. query) graph F is an m -tuple of data graphs $\mathcal{D}_F = (F_1, \dots, F_m)$, where $m \geq 1$, such that:

1. $F_i \subseteq F$, for $i = 1, \dots, m$, and
2. $\bigcup_i F_i = F$.

Each data (resp. query) graph F_i in a data (resp. query) graph decomposition is called a data (resp. query) graph segment. When, in a data/query graph decomposition, for all the pairs i, j , with $1 \leq i < j \leq m$, it also holds $F_i \cap F_j = \emptyset$, i.e. data (resp. query) graph segments are disjoint of each other, then the data (resp. query) graph decomposition is said to be non-redundant and the graph (resp. query) segments obtained form a partition of the triples of data (resp. query) graph F , called m -triple partition of F .

Notice that, $F_i \neq \emptyset$ for $i = 1, \dots, m$, since, because of Definitions 3.1.1 and 3.1.2, a data/query graph is nonempty.

It should be noted that, in a data or a query graph decomposition, a triple is (in general) allowed to participate in multiple data or query graph segments. At first sight, this redundancy seems to burden the system with the extra cost of storing and managing or evaluating more data. However, as it is shown in

subsequent chapters (see for example Chapter 6), if it is used appropriately it may lead to more efficient computation of the query answers, due to proper parallelization of the query execution.

Definition 3.3.2. Let $\mathcal{D}_F = (F_1, \dots, F_m)$, with $m \geq 1$, be a data (resp. query) graph decomposition of a data graph F , and F_i, F_j , with $i \neq j$, be two data (resp. query) graph segments in $\mathcal{D}_F$. A border node v of F_i and F_j , is a node that belongs to $\mathcal{N}(F_i) \cap \mathcal{N}(F_j) - L$. By $\mathcal{B}(F_i, F_j)$ we denote the set of border nodes of F_i and F_j , while, by $\mathcal{B}(F_i)$, we denote the set $\bigcup_{(1 \leq j \leq m) \wedge (j \neq i)} \mathcal{B}(F_i, F_j)$. Finally, by $\mathcal{B}(F)$ we denote the set of all border nodes of F i.e. $\mathcal{B}(F) = \bigcup_{1 \leq i \leq m} \mathcal{B}(F_i)$.

Notice that, according to the above definition, literals that occur in more than one segments in $\mathcal{D}_F$, are not considered to be border nodes.

Definition 3.3.3. Let $\mathcal{D}_Q = (Q_1, \dots, Q_n)$, where $n \geq 1$, be a query decomposition of a query graph Q . A node $n \in \mathcal{B}(Q)$ is said to be a common border node if $n \in \mathcal{B}(Q_i)$ for each Q_i in $\mathcal{D}_Q$. The set of common border nodes in Q is denoted as $\mathcal{CB}(Q)$.

Example 3.3.1. A data graph decomposition $\mathcal{D}_G$ (more specifically a 3-triple partition) of the data graph G of Fig. 3.1(a) appears in Fig. 3.3. The dark nodes correspond to the border nodes between the data graph segments, that is:

$$\begin{aligned}\mathcal{B}(G_1) &= \{\text{Person4}, \text{Article1}\} \\ \mathcal{B}(G_2) &= \{\text{Person4}, \text{Article1}, \text{Article2}\} \\ \mathcal{B}(G_3) &= \{\text{Article1}, \text{Article2}\}.\end{aligned}$$

A decomposition $\mathcal{D}_Q$ of a query Q into a 3 query graph segments (subqueries) Q_1, Q_2 , and Q_3 is illustrated in Fig. 3.4. The border nodes between the query graph segments are:

$$\begin{aligned}\mathcal{B}(Q_1) &= \{n1, n2\} = \{?P1, ?A\} \\ \mathcal{B}(Q_2) &= \{n2, n3\} = \{?A, ?P2\} \\ \mathcal{B}(Q_3) &= \{n1, n3\} = \{?P1, ?P2\}.\end{aligned}$$

while the set $\mathcal{CB}(Q)$ of common border nodes in Q is empty.

Notice that the query decomposition appearing in Fig. 3.4 is non-redundant.

Example 3.3.2. Consider the query graph Q appearing in the left part of Fig. 3.4. Q represents the query: “Find an article (variable $?A$) and its title (variable $?T$) published in *Journal1*, which has as authors a person (variable $?P1$) and his supervisor (variable $?P2$)”. It is easy to see that the evaluation of this query on the data graph G depicted in Fig. 3.3 returns the answers:

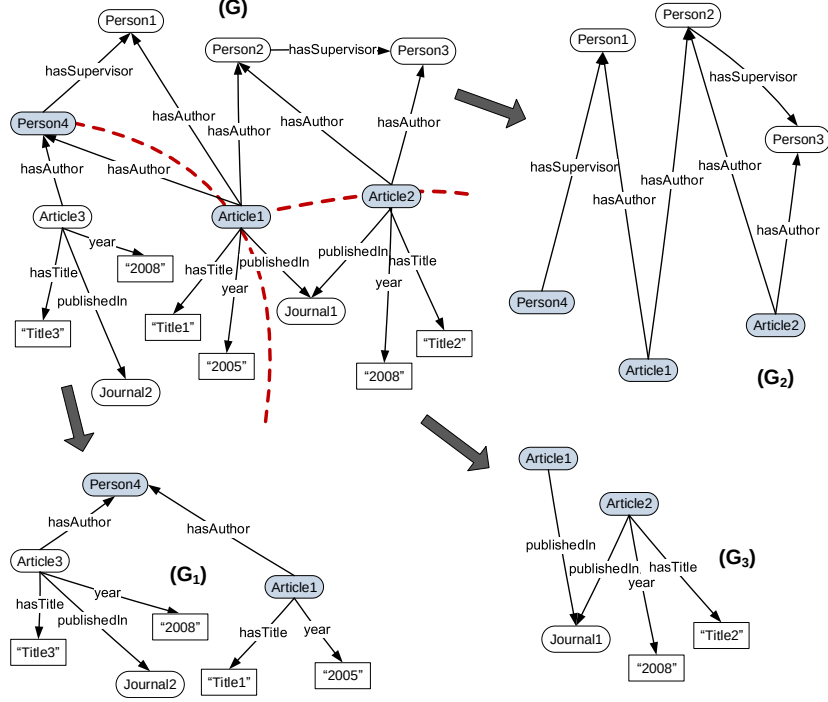


Figure 3.3: 3-triple partition of the data graph G of Fig. 3.1(a).

Answer 1: $(?P1, ?A, ?P2, ?T) = (Person4, Article1, Person1, "Title1")$.

Answer 2: $(?P1, ?A, ?P2, ?T) = (Person2, Article2, Person3, "Title2")$.

Notice, however, that, we cannot evaluate Q on a single data graph segment in $\mathcal{D}_G$ depicted in Fig. 3.3. Instead, all these graph segments are needed in order to compute the answers to this specific query Q as each of them contains part of the data needed to answer the query Q .

Query decomposition will be proved very useful in the subsequent Sections in query evaluation. The general idea behind the algorithms that will be presented is that, in order to find the answers to a query Q , it suffices to decompose Q into a tuple of subqueries $(Q_1, \dots, Q_m)$, find the answers (or partial answers) of $Q_1, \dots, Q_m$ and then combine appropriately these answers to construct the answers to the query Q .

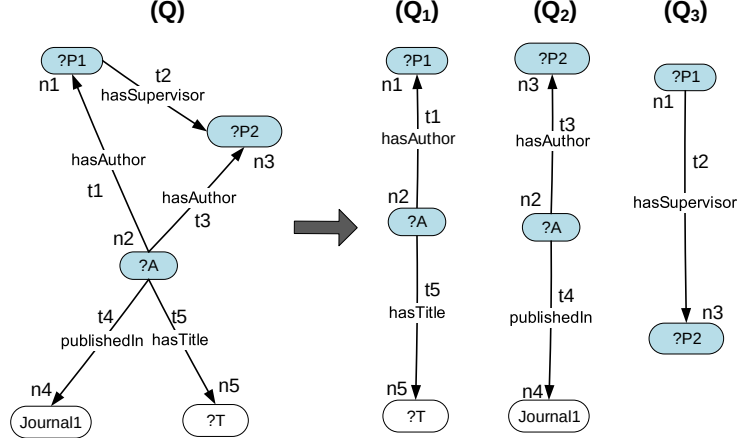


Figure 3.4: Query decomposition.

3.4 Partial embeddings

When a query Q is evaluated over a data graph segment G_i of a data graph G , it is likely that no embedding of Q in G_i exists. However, this does not necessarily mean that there is no embedding of Q in G , at all. Instead, it is possible that "part" of an embedding of Q in G has images in G_i , while other "parts" of the embeddings have images in other data graph segments of G . Then, to obtain the embedding of Q in G , we have to combine appropriately these "partial embeddings". This situation is formulated as follows:

Definition 3.4.1. A partial embedding of a query graph Q in a data graph G is a partial mapping $e : \mathcal{N}(Q) \rightarrow \mathcal{N}(G)$ such that for every node $v \in \mathcal{N}(Q)$ for which $e(v)$ is defined, the following properties hold:

1. if v is not a variable, then $e(v) = v$.
2. if v is a variable, then there exists a node $u \in \mathcal{N}(Q)$ for which $e(u)$ is defined and an edge label $p \in \mathcal{E}(Q)$, such that $(v, p, u) \in Q$ and $(e(v), p, e(u)) \in G$ or $(u, p, v) \in Q$ and $(e(u), p, e(v)) \in G$.

A partial embedding is said to be non-trivial if there exists a triple $(v_1, p, v_2) \in Q$ such that both $e(v_1)$ and $e(v_2)$ are defined and the triple $(e(v_1), p, e(v_2))$ belongs to G . In other words, a non-trivial partial embedding is a partial embedding that maps at least one edge of Q in G .

In essence, a partial embedding represents a mapping from a subset of nodes and edges of Q to a given data graph G . In other words, partial embeddings represent partial answers to Q , provided that, they can be appropriately “combined” with other “compatible” partial embeddings to give complete answers to the query Q .

The intuition behind Condition (2) is that when $e(v)$ is defined for a variable v of a query Q , then there is a triple t in Q such that the variable is either the subject or the object of t , and t is mapped, through e , to a triple in the data graph G . Notice that, as we will prove in the next Section, no answers are lost by imposing this condition to the definition of partial embeddings, while it substantially restricts the search space for computing partial embeddings.

It is easy to see that a total embedding e of Q in G is also a partial embedding of Q in G . Moreover, a total embedding of a subquery of Q , corresponds to a partial embedding of Q .

Definition 3.4.2. *Two partial mappings $e_1 : D_1 \rightarrow R_1$ and $e_2 : D_2 \rightarrow R_2$ are said to be compatible if for every node $v \in D_1 \cap D_2$ such that $e_1(v)$ and $e_2(v)$ are defined, it is $e_1(v) = e_2(v)$.*

Definition 3.4.3. *Let $e_1 : D_1 \rightarrow R_1$ and $e_2 : D_2 \rightarrow R_2$ be two compatible partial mappings. The join of e_1 and e_2 is the partial mapping $e : D_1 \cup D_2 \rightarrow R_1 \cup R_2$ defined as follows:*

$$e(v) = \begin{cases} e_1(v) & \text{if } e_1(v) \text{ is defined} \\ e_2(v) & \text{if } e_2(v) \text{ is defined and } e_1(v) \text{ is undefined} \\ \text{undefined} & \text{if both } e_1(v) \text{ and } e_2(v) \text{ are undefined} \end{cases}$$

Note that, the above definitions apply also to total embeddings as they are partial mappings. Notice also that in the first case of Definition 7.3.1, $e_2(v)$ may be defined or not. If it is defined, then the compatibility of the two partial mappings (embeddings) implies that $e_2(v) = e_1(v)$. It is trivial to prove that the join of two compatible partial embeddings is a partial embedding and that the join operation is commutative and associative. Therefore, we can refer to the partial embedding resulting by the join of n mutually compatible partial embeddings without ambiguity.

It should also be noted that if Q' is a subquery of a query Q and e is a total embedding of Q' in a data graph G , then e is a partial embedding of Q in G .

Example 3.4.1. *The mappings e_1 , with $e_1(?A) = \text{Article2}$ and $e_1(?T) = \text{Title2}$, and e_2 , with $e_2(?A) = \text{Article2}$ and $e_2(?W) = \text{Person3}$, are partial embeddings*

of the query graph of Fig. 3.1(b) in the data graph of Fig. 3.1(a). e_1 and e_2 are compatible and their join is the partial embedding e_3 , with $e_3(?A) = \text{Article2}$, $e_3(?T) = \text{Title2}$ and $e_3(?W) = \text{Person3}$.

3.5 Special forms of queries

In this Section we define several special forms of queries. Notice that queries of special forms are often easier to evaluate.

We begin by defining the *path queries*:

Definition 3.5.1. A query Q is said to be a path query of length n , with $n \geq 1$, if it is of the form $(v_0, p_1, v_1), (v_1, p_2, v_2), \dots, (v_{n-1}, p_n, v_n)$.

We now define the *generalized star queries* as follows:

Definition 3.5.2. A query Q is called a generalized star query if there exists a node $c \in \mathcal{N}(Q)$, called the central node of Q and denoted as $C(Q)$, such that for every triple $t = (u, p, v) \in Q$ it is either $u = c$ or $v = c$.

We now define three special forms of generalized star queries, namely *subject star queries*, *object star queries* and *subject-object star queries* (*s-query*, *o-query*, and *so-query* for sort respectively).

Definition 3.5.3. A generalized star query Q is said to be a subject star query (resp. object star query) if for every triple $t \in Q$ the central node c of Q is the subject (resp. object) of t .

Definition 3.5.4. A generalized star query Q is said to be a subject-object star query if for every triple $t \in Q$, the central node c of Q is either the subject or the object of t and there is a triple $t' \in Q$, such that c is the subject of t' .

The interest in the above special forms of queries lies in that these queries are easier to evaluate (are evaluated more efficiently and are amenable to parallelization) than the general query graphs. Besides, as we will see in the next Section, we can easily decompose a huge data graph into a set of graph segments such that a query (as those defined above) can be evaluated independently on each graph segment. These special classes of queries have the following property: for every query graph Q there exist a (non-redundant) decomposition into s-queries (or o-queries, or so-queries, or path queries of length 1). This follows trivially from the fact that every query that consists of a single triple belongs to each of these classes of queries.

The algorithms for query evaluation proposed in this Ph.D. thesis are based on these observations. In the following Sections we will use two of the above classes of queries, namely, generalized star queries and subject-object star queries (so-star queries).

Chapter 4

Query Evaluation by Joining Partial Embeddings

In this Chapter, we present a query evaluation algorithm, called Query Evaluation by Joining Partial Embeddings Algorithm, or QEJPE Algorithm for short, based on the idea of computing (possibly in a distributed manner) partial embeddings of (sub)queries and appropriately joining them in order to construct the answers of our query.

4.1 The QEJPE Algorithm

The query evaluation approach presented below, called QEJPE-algorithm, is based on the idea of computing (possibly in a distributed manner) partial embeddings of subqueries (query graph segments) of a query Q over data segments of a decomposed data graph G and combining these partial embeddings to obtain (total) embeddings of the initial query Q in the data graph G . To narrow down the search space for finding partial embeddings we introduce the concept of *useful partial embeddings*:

Definition 4.1.1. Let $\mathcal{D}_G = (G_1, \dots, G_m)$, with $m \geq 1$, be a data graph decomposition of a data graph G and let e be a partial embedding of a query graph Q in some G_i . Then e is called a *useful partial embedding* of Q in G_i if the following conditions hold:

1. e is non-trivial.

2. e is defined for all the nodes in $(\mathcal{N}(Q) \cap \mathcal{N}(G_i))$.
3. for each triple $(v, p, u) \in Q$, if $e(v)$ is defined and $e(v) \notin (\mathcal{B}(G_i) \cup L)$, then $e(u)$ is also defined and $(e(v), p, e(u))$ is a triple in G_i .
4. for each triple $(v, p, u) \in Q$, if $e(u)$ is defined and $e(u) \notin (\mathcal{B}(G_i) \cup L)$, then $e(v)$ is also defined and $(e(v), p, e(u))$ is a triple in G_i .

Notice that, according to the above definition, if v is a non-variable node of the query graph Q that maps to a non-border node of G_i , then the second property implies that $e(v)$ is defined, and the third and fourth properties enforce every triple that contains v to be mapped in G_i . More generally, the edges which start from or end to a node that maps to a non-border node in a data graph segment G_i should also have images that belong entirely to G_i otherwise the partial embedding cannot be used to construct a query answer.

Lemma 4.1.1. *Let $\mathcal{D}_G = (G_1, \dots, G_m)$, with $m \geq 1$, be a (redundant or non-redundant) data graph decomposition of a data graph G and let Q be a query graph. Then the following statements are equivalent:*

1. e is a total embedding of Q in G .
2. there exist mutually compatible useful partial embeddings $e_1, \dots, e_k$ of Q in $G_{i_1}, \dots, G_{i_k}$, respectively, for some $i_1, \dots, i_k$ with $1 \leq i_1 < \dots < i_k \leq m$, that satisfy the following properties:
 - (a) for every triple $(v, p, u) \in Q$ there exists some j for which $e_j(v)$, $e_j(u)$ are defined and $(e_j(v), p, e_j(u)) \in G_{i_j}$.
 - (b) the join of $e_1, \dots, e_k$ is e .

Proof. Assume that (1) holds, that is, e is an embedding of Q in G . Let $Q_i = \{(v, p, u) \in Q \mid (e(v), p, e(u)) \in G_i\}$, $1 \leq i \leq m$, and let I be the set of indices for which Q_i is non-empty, that is, $I = \{i \mid Q_i \neq \emptyset\}$. Since the query graph Q is non-empty, I must be also non-empty. Suppose that $|I| = k$ and let $i_1, \dots, i_k$ be the elements of I in increasing order.

For every j , $1 \leq j \leq k$, define the following mapping e_j from Q to G_{i_j} :

$$e_j(v) = \begin{cases} v & \text{if } v \text{ is a non-variable node in } \mathcal{N}(Q) \cap \mathcal{N}(G_{i_j}) \\ e(v) & \text{if } v \text{ is a variable node in } \mathcal{N}(Q_{i_j}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is not hard to see that e_j is a partial embedding of Q in G_{i_j} and that the join of $e_1, \dots, e_k$ is exactly e . Thus, property (2b) holds. In order to prove that property (2a) holds, consider a triple $(v, p, u) \in Q$. Then, $(v, p, u) \in Q_{i_j}$ for some j , which implies that $v, u \in \mathcal{N}(Q_{i_j})$ and $e(v), e(u) \in \mathcal{N}(G_{i_j})$. From the definition of e_j it follows that $e_j(v) = e(v)$ and $e_j(u) = e(u)$, which implies that $(e_j(v), p, e_j(u)) = (e(v), p, e(u))$, which is in G_{i_j} by the definition of Q_{i_j} .

It remains to prove that e_j is useful. The fact that e_j is non-trivial is straightforward, since Q_{i_j} is non empty. Moreover, it obviously satisfies condition (2) of Definition 4.1.1.

In order to prove that e_j satisfies condition (3) of Definition 4.1.1, consider a triple $(v, p, u) \in Q$ such that $e_j(v)$ is defined and $e_j(v) \notin (\mathcal{B}(G_{i_j}) \cup L)$. Since e is an embedding of Q in G , it must be $(e(v), p, e(u)) \in G$. Moreover, $e(v) = e_j(v)$ which implies that $e(v)$ is not a border node of G_{i_j} nor an element of L . Therefore, $e(v)$ appears only in G_{i_j} , which implies that $(e(v), p, e(u))$ must be a triple in G_{i_j} . Hence, $(v, p, u) \in Q_{i_j}$, which implies that $u \in \mathcal{N}(Q_{i_j})$ and by the definition of e_j it is $e_j(u) = e(u)$ (i.e. $e_j(u)$ is defined). The fact that $(e_j(v), p, e_j(u))$ is a triple in G_{i_j} is now clear, since it equals $(e(v), p, e(u))$. The proof for condition (4) of Definition 4.1.1, is similar.

For the other direction, assume that (2) holds. We first show that e (the join of $e_1, \dots, e_k$) is a total mapping from $\mathcal{N}(Q)$ to $\mathcal{N}(G)$. Notice that $e_1, \dots, e_k$ are compatible. Suppose that $v \in \mathcal{N}(Q)$, that is, v appears in some triple of the form (u, p, v) or (v, p, u) in Q . Then, $e_j(v)$ is defined for some j (by property (2a)), which implies (using the definition of join) that $e(v)$ is also defined.

We next show that e is an embedding of Q in G . Let v be a non-variable element in $\mathcal{N}(Q)$. From the definition of join, it must be $e(v) = e_j(v)$ for some j . Since e_j is a useful partial embedding, it is $e_j(v) = v$. Therefore, it holds $e(v) = v$.

Finally, consider a triple $(v, p, u) \in Q$. By property (2a), there exists some j such that $(e_j(v), p, e_j(u)) \in G_{i_j}$ which implies that $(e(v), p, e(u)) \in G$ (since $e(v) = e_j(v)$, $e(u) = e_j(u)$, and $G_{i_j} \subseteq G$). $\square$

Lemma 4.1.2. *Let $\mathcal{D}_Q = (Q_1, \dots, Q_n)$, with $n \geq 1$, be a query decomposition of a query graph Q and G be a data graph. Then e is a total embedding of Q in G if and only if there exist mutually compatible total embeddings $e_1, \dots, e_n$ of $Q_1, \dots, Q_n$ in G such that the join of $e_1, \dots, e_n$ is e .*

Proof. For the one direction, assume that e is a total embedding of Q in G . For every i define e_i to be the restriction of e in $\mathcal{N}(Q_i)$ (that is, $e_i : \mathcal{N}(Q_i) \rightarrow \mathcal{N}(G)$),

with $e_i(v) = e(v)$). Obviously e_i is a total mapping. Furthermore, for every non-variable element $v \in \mathcal{N}(Q_i)$ it is $e_i(v) = e(v) = v$ and for every triple $(v_1, p, v_2) \in Q_i$ it is $(e_i(v_1), p, e_i(v_2)) = (e(v_1), p, e(v_2)) \in G$, which implies that e_i is actually an embedding of Q_i in G .

Moreover, for every i, j with $i \neq j$, if $v \in \mathcal{N}(Q_i) \cap \mathcal{N}(Q_j)$ then it is $e_i(v) = e_j(v) = e(v)$, which implies that e_i and e_j are compatible. Therefore, the join e' of $e_1, \dots, e_n$ exists. It remains to show that $e' = e$. Consider an arbitrary $v \in \mathcal{N}(Q)$. Then v appears in some triple $t \in Q$. Since $\mathcal{D}_Q$ is a decomposition of Q , there exists some i such that $t \in Q_i$. Thus, $v \in \mathcal{N}(Q_i)$, which implies that $e_i(v)$ is defined. From the definition of join, it follows that $e'(v) = e_i(v)$, which implies $e'(v) = e(v)$.

For the other direction, assume that $e_1, \dots, e_n$ are compatible total embeddings of $Q_1, \dots, Q_n$ in G and let e be their join. Using the same argument as above, we can prove that for every $v \in \mathcal{N}(Q)$ there exists some i such that $e_i(v)$ is defined, which implies that $e(v)$ is also defined. Thus, e is a total mapping.

We next show that e is an embedding of Q in G . Consider any non-variable element $v \in \mathcal{N}(G)$. Since e is total, $e(v)$ is defined. From the definition of join, there must be some i such that $e(v) = e_i(v)$, which implies $e(v) = v$ (since e_i is an embedding).

Finally, let (v_1, p, v_2) be a triple in Q . Since $\mathcal{D}_Q$ is a decomposition of Q , (v_1, p, v_2) belongs to some Q_i . Since e_i is a total embedding of Q_i in G , it holds $(e_i(v_1), p, e_i(v_2)) \in G$, which implies $(e(v_1), p, e(v_2)) \in G$. $\square$

Theorem 4.1.1. *Let $\mathcal{D}_Q = (Q_1, \dots, Q_n)$, with $n \geq 1$, be a query decomposition of a query graph Q and $\mathcal{D}_G = (G_1, \dots, G_m)$, with $m \geq 1$, be a data graph decomposition of a data graph G . Then the following statements are equivalent:*

1. *e is a total embedding of Q in G .*
2. *for every j , with $1 \leq j \leq n$, there exist useful partial embeddings $e_{j,1}, \dots, e_{j,k_j}$ of Q_j in $G_{i_{j,1}}, \dots, G_{i_{j,k_j}}$ for some $i_{j,1}, \dots, i_{j,k_j}$ with $1 \leq i_{j,1} < \dots < i_{j,k_j} \leq m$ that satisfy the following properties:*
 - (a) *for every j , with $1 \leq j \leq n$, and every triple $(v, p, u) \in Q_j$ there exists some ℓ such that $e_{j,\ell}(v)$, $e_{j,\ell}(u)$ are defined and $(e_{j,\ell}(v), p, e_{j,\ell}(u)) \in G_{i_{j,\ell}}$.*
 - (b) *for every j_1, j_2, ℓ_1, ℓ_2 , with $1 \leq j_1 \leq j_2 \leq n$, $1 \leq \ell_1 \leq k_{j_1}$, $1 \leq \ell_2 \leq k_{j_2}$, the partial embeddings e_{j_1,ℓ_1} and e_{j_2,ℓ_2} are compatible.*

(c) the join of e_{j,ℓ_j} for all $j \in \{1, \dots, n\}$ and all $\ell_j \in \{1, \dots, k_j\}$ is e .

Proof. For the one direction, assume that (1) holds, that is, e is an embedding of Q in G . From Lemma 4.1.2 we conclude that there are mutually compatible total embeddings $e_1, \dots, e_n$ of $Q_1, \dots, Q_n$ in G , such that the join of $e_1, \dots, e_n$ is e . Now, from Lemma 4.1.1 we conclude that, for each Q_j , there exist mutually compatible useful partial embeddings $e_{j,1}, \dots, e_{j,k_j}$ of Q_j in $G_{i_{j,1}}, \dots, G_{i_{j,k_j}}$ such that property (a) holds and the join of $e_{j,1}, \dots, e_{j,k_j}$ is e_j . In order to show that property (b) holds, suppose for the sake of contradiction, that e_{j_1,ℓ_1} and e_{j_2,ℓ_2} are not compatible, for some j_1, j_2, ℓ_1, ℓ_2 . Then, there exists some v such that $e_{j_1,\ell_1}(v) \neq e_{j_2,\ell_2}(v)$. Since $e_{j_1}(v) = e_{j_1,\ell_1}(v)$ and $e_{j_2}(v) = e_{j_2,\ell_2}(v)$, the total embeddings e_{j_1} and e_{j_2} must also be incompatible, which is a contradiction. Therefore, property (b) holds. Finally, property (c) holds since for all j , the join of $e_{j,1}, \dots, e_{j,k_j}$ is e_j and the join of $e_1, \dots, e_n$ is e .

For the other direction, assume that (2) holds. From Lemma 4.1.1, it follows that for every j , the join e_j of the partial embeddings $e_{j,1}, \dots, e_{j,k_j}$ is a total embedding of Q_j in G . Moreover, the resulting embeddings $e_1, \dots, e_n$ are mutually compatible, since we have assumed that e_{j_1,ℓ_1} and e_{j_2,ℓ_2} are compatible for all j_1, j_2, ℓ_1, ℓ_2 . Now, from Lemma 4.1.2 it follows that the join e of $e_1, \dots, e_n$ is a total embedding of Q in G . $\square$

Theorem 4.1.1 implies a generic query evaluation strategy, named *Query Evaluation by Joining Partial Embeddings (QEJPE) strategy* consisting of four steps. The algorithm assumes an arbitrary decomposition of the data graph G into a tuple $\mathcal{D}_G$ of data graph segments $G_1, \dots, G_m$, with $m \geq 1$, stored into a cluster of computer nodes.

- Step 1:** Decompose the query Q into a tuple $\mathcal{D}_Q$ of subqueries $Q_1, \dots, Q_n$, with $n \geq 1$.
- Step 2:** Compute all possible useful partial embeddings of each subquery Q_j over each data graph segment G_i of G .
- Step 3:** For each subquery Q_j , collect all the partial embeddings of Q_j obtained in Step 2 and join them to get total embeddings of Q_j .
- Step 4:** To construct the total embeddings (i.e. answers) of Q , join the total embeddings obtained in Step 3 by using one embedding for each subquery, in all possible ways.

Notice that the above generic query evaluation strategy has several interesting properties: a) it is independent of the way the data graph is decomposed and the way the data graph segments obtained by this decomposition are stored in the nodes of the cluster, b) it is independent of the way the query graph is decomposed, and c) it is independent of the algorithm used to compute (partial) embeddings.

In Section 4.2, we present an implementation of this strategy on a cluster of commodity computers based on the MapReduce programming framework.

4.2 A MapReduce Implementation of the QE-JPE Algorithm

In this Section, we present a distributed algorithm implementing the query evaluation approaches presented in Section 4.1. The algorithm extends the query evaluation approach presented in Section 4.1 in order to take advantage of the commutation power provided by the MapReduce computation framework presented in Subsection 2.2.1.

The implementation of the proposed query evaluation algorithm in the MapReduce programming framework consists of *a preprocessing phase* followed by *two MapReduce phases*.

4.2.1 The Preprocessing Phase

The *preprocessing phase* is the first stage in the MapReduce implementation of the QEJPE Algorithm, where the setting is prepared. Note that a very similar preprocessing phase is also used in the MapReduce implementations of the eval-STARS and QE-with-Redundancy algorithms presented in Chapter 5 and Chapter 6 respectively.

In particular, the preprocessing phase accepts a query Q which is posed by the user and decomposes it into a tuple of subqueries $(Q_1, \dots, Q_n)$, with $n \geq 1$, following the decomposition principles determined by the specific query evaluation algorithm. These subqueries broadcasted or distributed to the mappers of the first MapReduce phase of the query evaluation algorithm. Preprocessing phase also constructs some auxiliary structures and emits them to the mappers/reducers that implement the algorithm. To define these structures we assume an enumeration $n_1, n_2, \dots, n_{|\mathcal{N}(Q)|}$ of the nodes of the query Q , so that $n_1, n_2, \dots, n_{|\mathcal{B}(Q)|}$ are

the border nodes of Q and $n_{|\mathcal{B}(Q)|+1}, \dots, n_{|\mathcal{N}(Q)|}$ are the non-border nodes of Q . We denote by I the function that gives the index of a node in $\mathcal{N}(Q)$ with respect to the above enumeration (that is, for every $x \in \mathcal{N}(Q)$ it holds $x = n_{I(x)}$). We also denote by I_{nb} the function from $\mathcal{N}(Q) - \mathcal{B}(Q)$ to $\{1, \dots, |\mathcal{N}(Q) - \mathcal{B}(Q)|\}$, with $I_{nb}(x) = I(x) - |\mathcal{B}(Q)|$. Similarly, we assume an enumeration $t_1, t_2, \dots, t_{|Q|}$ of the triples in Q .

Using the above enumeration functions, we now define the concept of *query prototype*. A query prototype is a triple of tuples of the form:

$$(BorderNodeFlags, NonBorderNodeFlags, TripleFlags)$$

where *BorderNodeFlags* is a tuple of $|\mathcal{B}(Q)|$ items, one item for each border node in $\mathcal{B}(Q)$. Similarly, the *NonBorderNodeFlags* is a tuple of $|\mathcal{N}(Q) - \mathcal{B}(Q)|$ items, one for each non border node in $\mathcal{N}(Q) - \mathcal{B}(Q)$. Finally, the tuple *TripleFlag* has $|Q|$ items, one for each triple in Q .

Consider now that a query prototype is assigned to each (sub)query Q_i . Each item in the tuples of the prototype has either the value '+' to denote the presence of the corresponding border node/non-border node/triple, in Q_i , or the value '-' to denote the absence of that node or triple.

We also construct a set¹ called *Missing Border Nodes (MBN)* as follows: $MBN = \{(b_i, Q_j) \mid b_i \in \mathcal{B}(Q) \text{ and } b_i \notin \mathcal{N}(Q_j)\}$. An element (b_i, Q_j) in MBN denotes that the border node b_i of Q does not appear among the nodes of the subquery Q_j of Q .

Based on the idea of query prototype we can represent a partial or total embedding e of a (sub)query in a similar way; i.e. as a triple of tuples of the form $(BorderNodeValues, NonBorderNodeValues, TriplesMatched)$. More specifically, *BorderNodeValues* stores the images of the border nodes of the query through the (partial)embedding, while *NonBorderNodeValues* stores the images of non border nodes of the query. The star symbol ('*') is placed in the corresponding node place if no image of that node is defined in e . Finally, *TriplesMatched* keeps track of the triples of the query that have images on the data graph through the (partial) embedding e (by putting a '+' sign or a '-' sign in the corresponding place of *TriplesMatched*).

In the preprocessing phase the users' query Q is decomposed into a tuple of subqueries $(Q_1, \dots, Q_n)$, with $n \geq 1$ and the auxiliary structures presented above are constructed. Preprocessing phase emits these structures with key the pair $(subqueryID, SegmentID)$ to the mappers of Phase 1, except of *MBN*

¹In the algorithms presented in this Section we represent the MBN set as list.

list, which is emitted directly to the reducers of Phase 1.

Example 4.2.1. Consider the query Q appearing in Fig. 3.4 and assume that the subqueries Q_1 , Q_2 , and Q_3 are constructed in the preprocessing phase. Assume also that the numbering functions has numbered the nodes and the edges of Q as shown in Fig. 3.4. Then, it is easy to see that $\mathcal{B}(Q) = \{n1, n2, n3\}$ while $\mathcal{N}(Q) - \mathcal{B}(Q) = \{n4, n5\}$. Finally, the lists of triples is $(t1, t2, t3, t4, t5)$. It is thus easy to see that the query prototypes for the subqueries Q_1 , Q_2 and Q_3 are:

$Q1: (<+,+,->, <-,+>, <+,-,-,+>)$

$Q2: (<-,+>, <+,->, <-,-,+>,+>)$

$Q3: (<+,-,+>, <-,->, <-,+,-,->)$

while the list of missing border nodes is $MBN = [(n1, Q2), (n2, Q3), (n3, Q1)]$.

The implementation of the QEJPE-algorithm consists of two map-reduce phases which follow the preprocessing phase :

1. In the first map-reduce phase the subqueries are applied to each graph segment, in isolation, and intermediate results are computed. More specifically, the mappers of phase 1 compute useful (partial or total) embeddings of the subqueries, by applying each subquery to each specific graph segment. Then the reducers of phase 1 combine (i.e. join) the partial embeddings to compute the total embeddings of each subquery. Notice that the total embeddings of the subqueries are, in general, partial embeddings of the query Q to the graph G .
2. In the second map-reduce phase, the embeddings of the subqueries are combined appropriately to produce the embedding of the query Q on the graph G . More specifically, the mapper of phase 2 fills the missing border nodes in each sub-query embeddings using the values obtained from the embeddings of the other subqueries. Then, reducers of phase 2 construct the embeddings of the query Q by combining compatible embeddings, one for each subquery.

4.2.2 Phase 1 of the QEJPE-algorithm

The *mapper of phase 1* gets as input a subquery Q_i and a graph segment G_j and evaluates Q_i on G_j obtaining in this way all useful (total and partial) embeddings. These embeddings are emitted to the reducers of Phase 1 with key the subquery ID Q_i . The procedure for the Mapper of Phase 1 is given below:

```

mapper1 (( $Q_i, G_j$ ), ( $G_jData, subqueryInfo$ ))
//( $Q_i, G_j$ ):  $Q_i$  is the ID of a subquery,  $G_j$  is the ID of a data segment
//  $G_jData$ : the content of the data graph segment  $G_j$ 
//  $SubqueryInfo$ : prototypes/border & non-border nodes/triples of  $Q$ 
begin
  compute  $E = \{e \mid e \text{ is a useful partial embedding of } Q_i \text{ in } G_jData\}$ ;
  for each  $e \in E$  do emit ( $[Q_i, e]$ );
end.

```

Example 4.2.2. (Continued from Example 4.2.1). Some embeddings of the subqueries Q_1 , Q_2 and Q_3 (see Fig. 3.4) in the segments G_1 , G_2 and G_3 (see Fig. 3.3) computed by the corresponding mappers and emitted with key the subquery ID, appear below. More specifically, a total embedding evaluated and emitted by the mapper working on (Q_1, G_1) is²:

(1) key = Q_1 , value = ($\langle Person4, Article1, * \rangle$, $\langle *, "Title1" \rangle$, $\langle +, -, -, + \rangle$)

The Mapper working on (Q_2, G_1) computes and emits the partial embedding:

(2) key = Q_2 , value = ($\langle *, Article1, Person4 \rangle$, $\langle *, * \rangle$, $\langle -, -, +, -, - \rangle$)

Among the embeddings obtained and emitted by the Mapper working on (Q_1, G_2) is the (partial) embedding:

(3) key = Q_1 , value = ($\langle Person2, Article2, * \rangle$, $\langle *, * \rangle$, $\langle +, -, -, - \rangle$)

Among the embeddings obtained and emitted by the Mapper working on (Q_2, G_2) are the (partial) embeddings:

(4) key = Q_2 , value = ($\langle *, Article1, Person1 \rangle$, $\langle *, * \rangle$, $\langle -, -, +, -, - \rangle$)

(5) key = Q_2 , value = ($\langle *, Article2, Person3 \rangle$, $\langle *, * \rangle$, $\langle -, -, +, -, - \rangle$)

The Mapper working on (Q_3, G_2) computes and emits the total embeddings:

(6) key = Q_3 , value = ($\langle Person4, *, Person1 \rangle$, $\langle *, * \rangle$, $\langle -, +, -, -, - \rangle$)

(7) key = Q_3 , value = ($\langle Person2, *, Person3 \rangle$, $\langle *, * \rangle$, $\langle -, +, -, -, - \rangle$)

The Mapper working on (Q_1, G_3) emits the partial embedding:

(8) key = Q_1 , value = ($\langle *, Article2, * \rangle$, $\langle *, "Title2" \rangle$, $\langle -, -, -, + \rangle$)

The Mapper working on (Q_2, G_3) emits the partial embeddings:

(9) key = Q_2 , value = ($\langle *, Article1, * \rangle$, $\langle Journal1, * \rangle$, $\langle -, -, -, +, - \rangle$)

(10) key = Q_2 , value = ($\langle *, Article2, * \rangle$, $\langle Journal1, * \rangle$, $\langle -, -, -, +, - \rangle$)

Finally, the mappers working on (Q_3, G_3) and (Q_3, G_1) return no (partial or total) embeddings.

²Notice that we can check if an embedding is total or partial by comparing it with the corresponding subquery prototype (see Example 4.2.1). An embedding is total if it has images for all (border and non-border) nodes and triples of the subquery (i.e. for all nodes and triples of the subquery the '+' sign appears in the corresponding place of the query prototype.)

It is important to note that the procedure for the **mapper1** does not determine a specific method for the computation of the useful (partial) embeddings of the subqueries. This means that, every algorithm that can compute all partial embeddings can be used in a specific implementation of the **mapper1**. Moreover, **mapper1** is independent of the way the data graph is stored.

A *Reducer of Phase 1* receives all useful (partial) embeddings of a subquery Q_i whose ID is the key of the reducer, in all graph segments $G_1, \dots, G_m$ of G . A reducer: (a) computes all total embeddings of Q_i in G and emits them to the mappers of Phase 2 with key the subquery ID, and (b) it finds all border node values from the total embeddings of Q_i that are missing from the total embeddings of other subqueries and emits them with the appropriate subquery IDs as keys. The reducer is defined as follows³:

```

reducer1( $Q_i, values$ )
//  $Q_i$ : a subquery ID.
//  $values$ : contains the list of the embeddings for  $Q_i$  and the MBN list
begin
  collect in a list  $F_i$  the total embeddings of  $Q_i$  appearing in values or
  obtained by joining compatible partial embeddings in values;
  if  $F_i$  is empty then EXIT; // there is no solution for the subquery  $Q_i$ 
                                // and thus for the original query  $Q$ 
  extract the MBN list from values;
  foreach embedding  $e = (bnv, nbnv, tm)$  in  $F_i$  do
    begin
      emit( $[Q_i, (bnv, nbnv)]$ ); // emits total embedding with key the subquery ID  $Q_i$ 
      for  $i = 1$  to  $|bnv|$  do
        if  $(bnv[i] \neq '*')$  then
          for each  $(n_i, Q_j)$  in MBN do
            emit( $[Q_j, (n_i, bnv[i])]$ );
        end
      end
    end
end.

```

Example 4.2.3. (Continued from Example 4.2.2). Among the total embeddings of Q_1 that constructs and emits reducer with key Q_1 are:

(1) $\Rightarrow$ key = Q_1 , value = ($\langle Person4, Article1, * \rangle$, $\langle *, "Title1" \rangle$)
 (3)+(8) $\Rightarrow$ key = Q_1 , value = ($\langle Person2, Article2, * \rangle$, $\langle *, "Title2" \rangle$)

³In the presentation of the procedures the following abbreviations are used: *bnv* stands for BorderNodeValues, *nbnv* for NonBorderNodeValues, and *tm* for TriplesMatched.

Taking into account the contents of the MBN list: $MBN = [(n1, Q2), (n2, Q3), (n3, Q1)]$ the reducer also emits the following missing border node values:

key = Q2, value = (1, Person2) key = Q2, value = (1, Person4)
key = Q3, value = (2, Article1) key = Q3, value = (2, Article2), ...

The Reducer for key Q2 constructs and emits the total embeddings for Q2:

(4)+(9)=> key = Q2, value = (<*, Article1, Person1>, <Journal1, *>)
(5)+(10)=> key = Q2, value = (<*, Article2, Person3>, <Journal1, *>)

and the following values for missing border nodes:

key = Q3, value = (2, Article1) key = Q3, value = (2, Article2)
key = Q1, value = (3, Person1) key = Q1, value = (3, Person3), ...

Reducer for key Q3 emits:

(6)=> key = Q3, value = (<Person4, *, Person1>, <*, *>)
(7)=> key = Q3, value = (<Person2, *, Person3>, <*, *>)
key = Q2, value = (1, Person2) key = Q2, value = (1, Person4)
key = Q1, value = (3, Person1) key = Q1, value = (3, Person3)

4.2.3 Phase 2 of the QEJPE-algorithm

Each Mapper in Phase2 manipulates the embeddings of a specific subquery. It fills in their missing border node values using values from the embeddings of other subqueries that have been emitted by the **reducer1** based in the information in MBN list and emits the resulted embeddings to the reducers of Phase 2 (the key is the tuple of the border node values). The mapper of Phase 2 is given below:

```
mapper2( $Q_i$ , values)
//  $Q_i$ : the ID of a subquery
// values: a list  $E$  of the parts (bnv, nbnv) of the total embeddings of  $Q_i$  and
// a list  $V$  of pairs ( $i, v$ ), where  $v$  is a candidate value for bnv[i]
begin
  for each embedding  $e = (bnv, nbnv)$  in  $E$  do
    for each instance  $bnv'$  of bnv using the values in  $V$  do
      emit([bnv', ( $Q_i$ , nbnv)]);
end.
```

Example 4.2.4. (Continued from Example 4.2.3). Mapper with key Q1 receives:

$E = [(<Person4, Article1, *>, <*, "Title1">), (<Person2, Article2, *>, <*, "Title2">), ...]$
 $V = [(3, Person1), (3, Person3), ...]$

This mapper produces instances of the border node tuples in E by replacing the '*' in the 3rd place with a value in V . Among the key-value pairs obtained and emitted in this way are:

$key = (Person4, Article1, Person1), value = (Q1, <*, "Title1">)$

$key = (Person2, Article2, Person3), value = (Q1, <*, "Title2">)$

The input of the Mapper with key $Q2$ is:

$E = [(<*, Article1, Person1>, <Journal1, *>),$
 $(<*, Article2, Person3>, <Journal1, *>), \dots]$

$V = [(1, Person2), (1, Person4), \dots]$

Some of the instances that this mapper produces and emits are:

$key = (Person4, Article1, Person1), value = (Q2, <Journal1, *>)$

$key = (Person2, Article2, Person3), value = (Q2, <Journal1, *>)$

The input of the Mapper with key $Q3$ is:

$E = [(<Person4, *, Person1>, <*, *>), (<Person2, *, Person3>, <*, *>)]$
 $V = [(2, Article1), (2, Article2)]$

Some key-value pairs produced and emitted (as above) by this mapper are:

$key = (Person4, Article1, Person1), value = (Q3, <*, *>)$

$key = (Person2, Article2, Person3), value = (Q3, <*, *>)$

In each reducer of phase 2, the embeddings (one for each subquery in $(Q_1, \dots, Q_n)$) are joined⁴ to construct the final answers of Q . The reducer of phase 2 is given below:

```

reducer2(key, values)
// key: a tuple of border node values
// values: pairs of the form  $(Q_i, \text{partial embedding for non-border nodes})$ 
begin
  for each join of compatible embeddings
    obtained by using one embedding for each subquery do
      Emit the result produced by this join;
end.

```

Example 4.2.5. (Continued from Example 4.2.4). The Reducer with key $(Person4, Article1, Person1)$ receives the list:

$[(Q1, <*, "Title1">), (Q2, <Journal1, *>), (Q3, <*, *>)]$

Combining (i.e. joining) these embeddings the reducer returns the answer:

$<Person4, Article1, Person1, Journal1, "Title1">$

The reducer with key $(Person2, Article2, Person3)$ receives the following list:

$[(Q1, <*, "Title2">), (Q2, <Journal1, *>), (Q3, <*, *>)]$

which joins giving the answer:

$<Person2, Article2, Person3, Journal1, "Title2">$

⁴Notice that the joined embeddings are, by construction, compatible.

Notice that no other reducer returns solution (as they do not receive embeddings for all subqueries). This can be verified by considering all possible embeddings of all subqueries, which do not appear, for space reasons, in the examples of this Subsection.

4.3 Discussion

QEJPE-algorithm computes the answers to the given query Q correctly, independently of a) the data graph partitioning, b) the way the graph segments are stored, c) the query graph decomposition, d) the algorithm used for calculating intermediate (partial) results.

Some improvements to the proposed algorithm are as follows :

1. In order to obtain all total embeddings of a subquery Q_i in **reducer1** it suffices to combine partial embeddings obtained from different data graph segments. However, such provenance information is not emitted from **mapper1** in its present form. It is, however, easy to adapt the QEJPE-algorithm, so as the **mapper1** emits this information to **reducer1** and the **reducer1** takes it into account to construct more efficiently total embeddings of the subqueries.
2. In Example 4.2.3, several instances of **reducer1** may emit the same values, either embeddings or missing node values, to the **mapper2**. This is due to the fact that the same embedding or the same candidate missing node value may be found and emitted by several reducers. Thus, a specific instance of **mapper2** may receive multiple times the same value which may lead in the construction of the same embedding several times. A possible optimization is to eliminate redundant values from the lists E of embeddings and the list V of candidate values for missing nodes that an instance of a **mapper2** receives before the computation of embeddings that will be emitted to **reducer2**.

Chapter 5

Query Evaluation by Decomposing Queries Into Generalized Stars

In this Chapter, we present a query evaluation algorithm, called eval-STARS Algorithm, based on the idea of decomposing the given query Q into subqueries of the form of generalized star, computing (possibly in a distributed manner) the answers of these subqueries, and appropriately joining these answers to construct the answers of the initial query Q .

5.1 The eval-STARS Algorithm

In this Section we present another approach, called eval-STARS algorithm, for evaluating queries over linked data. The algorithm is based on assumptions similar to these on which the QEJPE-algorithm, presented in Chapter 4, is based. The main difference is that we now impose subqueries obtained from the decomposition of a user query Q to be in the form called *generalized star queries*. Besides, the algorithm based on evaluation of total embeddings of the subqueries instead of partial embeddings.

As we proved in Lemma 4.1.2, in order to compute the answers to a given query in a data graph G , we can decompose the query into a tuple of subqueries, compute the embeddings of the subqueries in G (which may be a more efficient

task due to the simpler or special form of the subqueries) and then join these embedding to obtain the desired result. However, given a target class of queries C , it may not be always possible to decompose an arbitrary query Q into subqueries that belong to C . For example, if C is the class of path queries of length 3, in other words if the subqueries must be of the form $\{(u, p, v), (v, p', w), (w, p'', z)\}$, then it can be proved that it is not possible to decompose every user query into a set of path queries of length 3. Nevertheless, if the target class C is the class of generalized star queries, then for every query Q there exist a (non-redundant) decomposition of Q into a tuple of generalized star subqueries. This trivially follows from the fact that every query that consists of a single triple is also a generalized star query (with either the subject or the object being the central node). We next present a more general result, relating the decomposition of a query graph Q into generalized star subqueries, to the *node covers* of this query graph.

Definition 5.1.1. *Let Q be a query graph. A set of nodes $V \subseteq \mathcal{N}(Q) - L$ is called a node cover of Q if for every triple $(s, p, o) \in Q$, it holds either $s \in V$ or $o \in V$.*

Lemma 5.1.1. *Let Q be a query graph and $V = \{v_1, \dots, v_k\}$ be a node cover of Q . For each $v_i \in V$ define the generalized star query $Q_{v_i} = \{t \in Q \mid t = (s, p, v_i)\} \cup \{t \in Q \mid t = (v_i, p, o) \text{ and } o \notin V\}$. Then $\mathcal{D}_Q = (Q_{v_1}, \dots, Q_{v_k})$ is a non-redundant decomposition of Q .*

Proof. It easy to see that $(Q_{v_1}, \dots, Q_{v_k})$ forms a decomposition of Q since:

- (1) by construction $Q_{v_i} \subseteq Q$, for $i = 1, \dots, k$, and
- (2) $\bigcup_i Q_{v_i} = Q$, since for every triple $t = (s, p, o) \in Q$, either $s \in V$ or $o \in V$. If $o \in V$ then, by construction $t \in Q_o$. Otherwise (i.e. if $s \in V$ and $o \notin V$) then $t \in Q_s$.

We will now prove (by contradiction) that $\mathcal{D}_Q = (Q_{v_1}, \dots, Q_{v_k})$ is non-redundant. Assume that $\mathcal{D}_Q$ is redundant. Then there exists a triple $t = (s, p, o) \in Q$ such that t belongs to two different subqueries in $\mathcal{D}_Q$. It is easy to see that these subqueries should be Q_s and Q_o and $s, o \in V$. However, since $t \in Q_s$, then, by construction, $o \notin V$, which contradicts with the fact that $s, o \in V$. $\square$

Therefore, if a set of nodes is a node cover of a query Q , then its elements are the central nodes of the generalized star subqueries in a non-redundant decomposition of Q . It turns out that the converse also holds.

Lemma 5.1.2. Let Q be a query graph, let $\mathcal{D}_Q = (Q_1, \dots, Q_k)$ be a decomposition of Q such that $Q_1, \dots, Q_k$ are generalized star queries and let $c_1, \dots, c_k$ be their central nodes. Then $\{c_1, \dots, c_k\}$ is a node cover of Q .

Proof. It immediately follows from Definitions 3.3.1, 3.5.2 and 5.1.1. $\square$

Example 5.1.1. In Fig. 5.1 we see a decomposition of the query Q into three generalized star queries Q_1 , Q_2 and Q_3 , which is obtained by the construction of Lemma 5.1.1, using the node cover $\{n_4, n_2, n_5\}$ of Q .

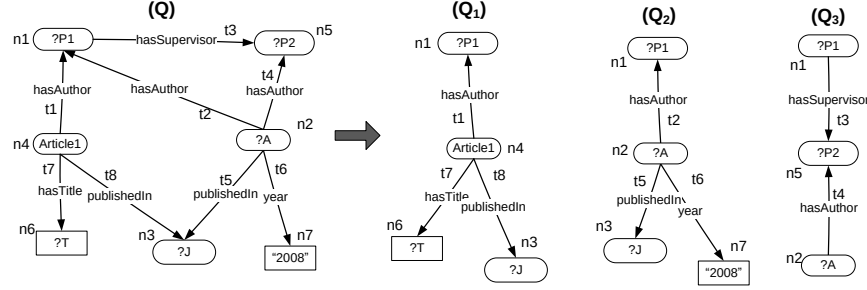


Figure 5.1: Query decomposition into star queries.

Following the discussion above we can specialize the generic query evaluation strategy (QEJPE strategy) presented in Section 4.1, obtaining in this way a new algorithm called *eval-STARs algorithm*. As in the case of QEJPE strategy we assume an arbitrary decomposition of the data graph G into a tuple $\mathcal{D}_G$ of data graph segments $G_1, \dots, G_m$, with $m \geq 1$, stored into a cluster of computer nodes.

The *eval-STARs algorithm* consists of the following steps:

- Step 1:** Decompose the query Q into a tuple of generalized star subqueries $\mathcal{D}_Q = (Q_1, \dots, Q_n)$, with $n \geq 1$.
- Step 2:** Compute all possible embeddings of each triple in Q over each data graph segment G_i of G .
- Step 3:** For each subquery Q_j , collect the embeddings of all the triples in Q_j and join compatible embeddings in all possible ways to compute the total embeddings of Q_j in G .
- Step 4:** To construct the total embeddings (i.e. answers) of Q , join the total embeddings obtained in Step 3 by using one embedding for each subquery, in all possible ways.

Note that eval-STARS algorithm applies two query decomposition processes. Initially, the given query is decomposed into generalized star queries (Step 1 of eval-STARS) and each star query is further decomposed (Step 2 of eval-STARS) into its triples. On the contrary, QEJPE applies a single decomposition (Step 1 of QEJPE). Following this stepwise approach of two decompositions, in fact, we achieve the construction of the total embeddings in two phases, where each phase gathers the compatible partial embeddings and join them together (i.e., it applies Lemma 4.1.2 twice). Although this extra decomposition could be thought of as a redundant step, in parallel computation (see Subsection 2.2.1), such an approach brings a significant performance improvement and facilitates the distribution of both the intermediate data and the computation.

5.2 A Map-Reduce Implementation of the eval-STARS Algorithm

In this Section we present a MapReduce based implementation of the eval-STARS query evaluation algorithm presented in Section 5.1. The algorithm is based on similar assumptions on which the QEJPE-algorithm is based. The main difference is that in eval-STARS algorithm, a queries Q posed by the user is decomposed into a tuple of queries $(Q_1, \dots, Q_n)$, with $n \geq 1$, of a specific form called *generalized star queries*. The query decomposition may be redundant or non-redundant.

The implementation of the algorithm consists of a preprocessing phase followed by two map-reduce phases: The first map-reduce phase takes advantage of the generalized star form of the sub-queries and focuses on evaluating the generalized star subqueries over the input segments. The results of the sub-queries are emitted to the second phase, which combines them properly in order to produce the answers of the initial query.

5.2.1 Preprocessing Phase

In the *preprocessing phase* the users' query Q is decomposed into a tuple of generalized star subqueries, with $n \geq 1$ and the auxiliary structures presented in Subsection 4.2.1 are constructed. Preprocessing phase emits these structures with key the pair $(subqueryID, SegmentID)$ to the mappers of Phase 1.

Example 5.2.1. Consider the query graph appearing in the left part of Fig. 5.1 which is decomposed into three generalized star subqueries Q_1 , Q_2 , and Q_3 appearing in the right part of Fig. 5.1. The border nodes are $\mathcal{B}(Q) = \{n1, n2, n3\}$, while the non-border nodes are $\mathcal{N}(Q) - \mathcal{B}(Q) = \{n4, n5, n6, n7\}$.

The query prototypes are the following:

$Q_1: (<+, -, +>, <+, -, +, ->, <+, -, -, -, -, +, +>)$

$Q_2: (<+, +, +>, <-, -, -, +>, <-, +, -, -, +, +, -, ->)$

$Q_3: (<+, +, -, ->, <-, +, -, ->, <-, -, +, +, -, -, ->)$

The list $MBN = [(n_2, Q_1), (n_3, Q_3)]$ is also constructed in preprocessing phase.

5.2.2 Phase 1 of the algorithm

The first phase of the algorithm computes the embeddings of the generalized star subqueries $Q_1, \dots, Q_n$ in G .

In Phase 1 each *mapper* gets as input a generalized star subquery Q_i , a graph segment G_j and the *MBN* list. Let $c_i = C(Q_i)$ be the central node of Q_i (recall that this node appears in every triple of Q_i). The operation of the mapper is divided into two parts.

Part 1: The mapper computes the embeddings of each triple of Q_i in G_j that map the central node c_i to a *border node* or to a literal, and emits the results to appropriate reducers. More specifically, let $t = (s, p, o)$ be a triple that belongs to subquery Q_i and let e be an embedding of t into G_j such that $e(c_i) \in \mathcal{B}(G_j)$. If the central node of Q_i is s then the mapper emits a pair $(key, value)$, where $key = (Q_i, e(s))$ and $value = (o, e(o))$. Otherwise (i.e., if the central node of Q_i is o) then $key = (Q_i, e(o))$ and $value = (s, e(s))$.

Notice that embeddings of triples in Q_i that map c_i to different nodes of G_j are incompatible and cannot be joined to obtain an embedding of Q_i . Since the value of c_i is included in the key, incompatible embeddings of triples are emitted to different reducers, while compatible embeddings are emitted to the same reducer.

Part 2: This part of the **mapper1** computes all embeddings of Q_i into G_j which map the central node of Q_i to a non-border and non-literal node of G_j . Notice that if for some embedding e of Q_i in G the value of c_i is a non-border and non-literal node of G_j (i.e., is $e(c_i) \in (\mathcal{N}(G_i) - (\mathcal{B}(G_i) \cup L))$), then $e(v) \in G_j$ for every node $v \in \mathcal{N}(Q_i)$. This means that e is an embedding of Q_i into G_j and it can be computed locally i.e. no other data graph segments are needed to compute e .

The computation of the embeddings of Q_i into G_j , which map c_i to a non-border node of G_j can be achieved either by adding an appropriate conjunct to Q_i , or by computing all the embeddings of Q_i in G_j and then removing those that assign border nodes to c_i . The embeddings computed in the second part of the mapper are directly emitted to the mappers of Phase 2 (rather than to the reducers of Phase 1). Similarly, the values of missing border nodes are emitted to the mappers of Phase 2.

```

mapper1(( $Q_i$ ,  $G_j$ ), ( $G_j$ Data,  $B(G_j$ Data),  $subqueryInfo$ ,  $MBN$ ))
//( $Q_i, G_j$ ):  $Q_i$  is the ID of a subquery;  $G_j$  is the ID of a data segment
//  $G_j$ Data: the content of the data graph segment  $G_j$ 
//  $B(G_j$ Data): the set of border nodes of  $G_j$ 
//  $SubqueryInfo$ : prototypes/border & non-border nodes/triples of  $Q$ 
//  $MBN$ : the list of missing border nodes
begin
  Let  $c_i = C(Q_i)$ ;
  % Part 1
  foreach triple  $t = (c_i, p, o)$  in  $Q_i$  do
    begin
      compute  $E = \{e \mid e \text{ is an embedding of } t \text{ in } G_j\text{Data and } e(c_i) \in B(G_j\text{Data}) \}$ ;
      foreach embedding  $e$  in  $E$  do
        emit([(Qi, e(ci)), (o, e(o))]);
      end
    foreach triple  $t = (s, p, c_i)$  in  $Q_i$  do
      begin
        compute  $E = \{e \mid e \text{ is an embedding of } t \text{ in } G_j\text{Data and } e(c_i) \in (B(G_j\text{Data}) \cup L) \}$ ;
        for each embedding  $e$  in  $E$  do
          emit([(Qi, e(ci)), (s, e(s))]);
        end
      % Part 2
      compute  $E = \{e \mid e \text{ is a embedding of } Q_i \text{ in } G_j\text{Data and } e(c_i) \notin (B(G_j\text{Data}) \cup L) \}$ ;
      for each embedding  $e = (bn, nbv)$  in  $E$  do
        begin
          emitToSecondPhase([(Qi, (bnv, nbv))]); // i.e. to the mapper of phase 2
          for  $k = 1$  to  $|nbv|$  do
            if ( $nbv[k] \neq \text{'*'} \text{'}$ ) then
              for each ( $n_k, Q_j$ ) in  $MBN$  do
                emitToSecondPhase([(Qj, (nk, nbv[k]))]);
            end
          end
        end
      end

```

Example 5.2.2. (Continued from Example 5.2.1). In this example, we assume that the query graph Q and its generalized star subqueries are those appearing in Fig. 5.1, while the data graph G and the graph segments obtained by decomposing G are those appearing in Fig. 3.3. Below, we see the application of **mapper1** on the pairs of subqueries and graph segments:

Applying **mapper1** on (Q_1, G_1) results in emission (see Part 1 of the procedure for **mapper1**) of the following (key,value) pairs to the **reducer1**:

key = (Q1, Article1), value = (n1, Person4) (embedding of t1)

key = (Q1, Article1), value = (n6, "Title1") (embedding of t7)

No key value pairs are emitted to Phase 2 (see Part 2 of the procedure for **mapper1**).

Applying **mapper1** on (Q_2, G_1) results in emission (see Part 1) of the following key,value pair to the **reducer1**:

key = (Q2, Article1), value = (n1, Person4) (embedding of t2)

Besides, the following key,value pairs are emitted directly to the **mapper2** (Mapper of Phase 2) (see Part 2):

key = Q2, value = (<Person4,Article3,Journal2>, <*,*,*, "2008">)

key = Q1, value = (n2, Article3)

key = Q3, value = (n3, Journal2)

Notice that the last two emissions are conducted by the MBN list which, as we have seen in Example 5.2.1, is $MBN = [(n_2, Q_1), (n_3, Q_3)]$.

Applying **mapper1** on (Q_3, G_1) results in emission (see Part 1) of the following (key,value) pairs to **reducer1**:

key = (Q3, Person4), value = (n2, Article1) (embedding of t4)

key = (Q3, Person4), value = (n2, Article3) (embedding of t4)

No key value pairs are emitted to Phase 2.

Applying **mapper1** on (Q_1, G_2) results in emission (see Part 1) of the following (key,value) pairs to **reducer1**:

key = (Q1, Article1), value = (n1, Person1) (embedding of t1)

key = (Q1, Article1), value = (n1, Person2) (embedding of t1)

No key value pairs are emitted to Phase 2.

Applying **mapper1** on (Q_2, G_2) results in emission (see Part 1) of the following key,value pairs to the **reducer1**:

key = (Q2, Article1), value = (n1, Person1) (embedding of t2)

key = (Q2, Article1), value = (n1, Person2) (embedding of t2)

key = (Q2, Article2), value = (n1, Person2) (embedding of t2)

key = (Q2, Article2), value = (n1, Person3) (embedding of t2)

No key value pairs are emitted to Phase 2.

Applying **mapper1** on (Q_3, G_2) results in no emission of any (key, value) pair to **reducer1** (see Part1). However, the following (key, value) pairs are emitted (see Part 2) to **mapper2**:

key = Q3, value = ($\langle \text{Person4}, \text{Article1}, * \rangle, \langle *, \text{Person1}, *, * \rangle$)
key = Q3, value = ($\langle \text{Person2}, \text{Article2}, * \rangle, \langle *, \text{Person3}, *, * \rangle$)
key = Q1, value = (n2, Article1)
key = Q1, value = (n2, Article2)

Applying **mapper1** on (Q_1, G_3) results in emission of the following (key, value) pair to **reducer1**:

key = (Q1, Article1), value = (n3, Journal1) (t8)

No key value pairs are emitted to Phase 2 (see Part2).

Applying **mapper1** on (Q_2, G_3) results in emission of the following (key, value) pair to **reducer1**:

key = (Q2, Article1), value = (n3, Journal1) (t5)
key = (Q2, Article2), value = (n3, Journal1) (t5)
key = (Q2, Article2), value = (n7, "2008") (t6)

No key value pairs are emitted to Phase 2.

Applying **mapper1** on (Q_3, G_3) results in no emission of any (key, value) pair.

Concerning the Reducer of Phase 1 For each key (Q_i, v) the corresponding reducer computes all the embeddings of Q_i that map the central node c_i of Q_i to v . The input to this reducer is a list of pairs of the form (n_k, u) , where n_k is a node of Q_i different from c_i and u is a possible value for n_k in an embedding of Q_i in G . Suppose that $n_{k_1}, n_{k_2}, \dots, n_{k_m}$ are the non-central nodes in Q_i . Then, for every $j = 1, \dots, m$, the reducer constructs a set $L[k_j]$ of all possible values for node n_{k_j} . More specifically, for each element $(x_1, x_2, \dots, x_m)$ of the cartesian product $L[k_1] \times L[k_2] \times \dots \times L[k_m]$, it constructs an embedding $e = (\text{bnv}, \text{nbv})$ of Q_i in G , such that $e(c_i) = v$ and $e(n_{k_j}) = x_j$ and emits $(Q_i, (\text{bnv}, \text{nbv}))$ (see Subsection 4.2.1 for the representation of an embedding). Moreover, if at least one embedding of Q_i has been found, Reducer 1 emits the values of missing border nodes.

reducer1((Q_i, v), values)

// Q_i : a subquery ID

// v : the value of the central node of Q_i

// values: contains (i) a list of pairs (x, u) , with $x \neq C(Q_i)$ and

u is a candidate image of x and (ii) the MBN list.

```

begin
% Part 1
  allNonEmpty = true;
  foreach non-central node  $x$  in  $Q_i$  do
    begin
       $L[I(x)] = \{u \mid (x, u) \in \text{values}\};$ 
      if  $L[I(x)] = \emptyset$  then allNonEmpty = false;
    end
% Part 2
  if allNonEmpty = true then // i.e. there are values for all non-central nodes of  $Q_i$ 
    begin
      create an embedding with undefined values;
       $(\text{bnv}, \text{nbnv}) = (\langle *, \dots, * \rangle, \langle *, \dots, * \rangle);$ 
       $c_i = C(Q_i);$ 
       $L[I(c_i)] = \{v\};$ 
      if  $c_i$  is a border node then
         $\text{bnv}[I(c_i)] = v;$ 
      else  $\text{nbnv}[I_{nb}(c_i)] = v;$ 
       $E = \{(\text{bnv}, \text{nbnv})\};$ 
      for each non-central node  $x$  in  $Q_i$  do
        begin
           $E' = \emptyset;$ 
          foreach  $e$  in  $E$  do
            foreach  $u$  in  $L[I(x)]$  do
              begin
                create a copy  $e' = (\text{bnv}', \text{nbnv}')$  of  $e$ ;
                if  $x$  is a border node then
                   $\text{bnv}'[I(x)] = u;$ 
                else  $\text{nbnv}'[I_{nb}(x)] = u;$ 
                insert  $(\text{bnv}', \text{nbnv}')$  in  $E'$ ;
              end
            end
           $E = E';$ 
        end
      foreach embedding  $e = (\text{bnv}, \text{nbnv})$  in  $E$  do
        emit  $([Q_i, (\text{bnv}, \text{nbnv})]);$ 
      foreach  $(x, Q_j)$  in  $MBN$  do
        if  $x$  is a node in  $Q_i$  then
          foreach  $u$  in  $L[I(x)]$  do emit  $([Q_j, (x, u)]);$ 
        end
    end
  end.

```

Example 5.2.3. (Continued from Example 5.2.2).

The reducer with key $(Q_1, \text{Article1})$ receives the following list of values:

$[(n_1, \text{Person4}), (n_6, \text{"Title1"}), (n_1, \text{Person1}), (n_1, \text{Person2}), (n_3, \text{Journal1})]$.

Notice that, as we can conclude from the sub-query prototypes appearing in Example 5.2.1, the border nodes of Q is $\mathcal{B}(Q) = \{n_1, n_2, n_3\}$, while the non-border are $\mathcal{N}(Q) - \mathcal{B}(Q) = \{n_4, n_5, n_6, n_7\}$. Besides, from Fig. 5.1, we see that the central node of Q_1 is n_4 while its non-central nodes are n_1, n_3 and n_6 . Finally, the MBN list is $\text{MBN} = [(n_2, Q_1), (n_3, Q_3)]$. Taking into account the above, the **reducer1** with key $(Q_1, \text{Article1})$, concludes by applying Part1 of the procedure that it has received values for all non-central nodes of Q_1 . More specifically, **reducer1** constructs the following lists:

$L[1] = [\text{Person1}, \text{Person2}, \text{Person4}]$

$L[3] = [\text{Journal1}]$

$L[6] = [\text{"Title1"}]$

which contain the values for the non-central nodes n_1, n_3 and n_6 respectively. Combining these values, as well as the value Article1 of the central node n_4 , **reducer1** in Part 2 constructs and emits the following (key,value) pairs (that represent embeddings of Q_1):

key = Q_1 , value = $(\langle \text{Person1}, *, \text{Journal1} \rangle, \langle \text{Article1}, *, \text{Title1}, *_i \rangle)$

key = Q_1 , value = $(\langle \text{Person2}, *, \text{Journal1} \rangle, \langle \text{Article1}, *, \text{Title1}, *_i \rangle)$

key = Q_1 , value = $(\langle \text{Person4}, *, \text{Journal1} \rangle, \langle \text{Article1}, *, \text{Title1}, *_i \rangle)$

Besides, **reducer1**, based on the MBN list, emits the following:

$Q_3, (n_3, \text{Journal1})$

The reducer with key $(Q_2, \text{Article1})$ receives the following list of values:

$[(n_1, \text{Person4}), (n_1, \text{Person1}), (n_1, \text{Person2}), (n_3, \text{Journal1})]$.

Based on this values it constructs the following lists (corresponding to the values of the non-central nodes n_1, n_3 and n_7 of the subquery Q_2):

$L[1] = [\text{Person1}, \text{Person2}, \text{Person4}]$

$L[3] = [\text{Journal1}]$

$L[7] = []$

From the above we see that the list for the non-central node n_7 is empty. Thus, these values cannot be used to construct a valid embedding for the query Q_2 . Therefore, nothing is emitted to the next phase from this reducer.

The reducer with key $(Q_2, \text{Article2})$ receives the following list of values:

$[(n_1, \text{Person2}), (n_1, \text{Person3}), (n_3, \text{Journal1}), (n_7, \text{"2008"})]$.

It constructs the lists:

$L[1] = [\text{Person2}, \text{Person3}]$

$L[3] = [Journal1]$

$L[7] = ["2008"]$

Combining these values, as well as the value *Article2* of the central node n_2 , Part 2 of **reducer1** constructs and emits the following (key,value) pairs (that represent embeddings of Q_2):

key = Q_2 , value = ($\langle Person2, Article2, Journal1 \rangle$, $\langle *, *, *, "2008" \rangle$)

key = Q_2 , value = ($\langle Person3, Article2, Journal1 \rangle$, $\langle *, *, *, "2008" \rangle$)

Besides, **reducer1**, based on the MBN list, emits the following:

key = Q_1 , value = ($n_2, Article2$)

key = Q_3 , value = ($n_3, Journal1$)

The reducer with key ($Q_3, Person4$) receives the following list of values:

$[(n_2, Article1), (n_2, Article3)]$.

Based on this values it constructs the following lists (corresponding to the values of the non-central nodes n_1 and n_2 of the subquery Q_3):

$L[1] = []$

$L[2] = [Article1, Article3]$

From the above we see that the list for the non-central node n_1 is empty. Thus, these values cannot be used to construct a valid embedding for the query Q_3 . Therefore, nothing is emitted to the next phase from this reducer.

5.2.3 Phase 2 of the algorithm

Phase 2 of the algorithm is similar to the Phase 2 of the QEJPE-algorithm presented in Subsection 4.2.3. The input of each *mapper* of Phase 2, consist of all the embeddings of a specific subquery Q_i . Besides, for each border node that does not occur in Q_i , mapper gets as input, the values assigned to this node by the embeddings of the other queries. These values are sent by the mappers and reducers of Phase 1 based on the MBN list. Based on its input, the mappers of Phase 2 fills in their missing border node values using the corresponding input values, and emits the resulted embeddings to the reducers of Phase 2 using as key the tuple of the border node values. This means that two embeddings are emitted to the same reducer if and only if they are compatible.

mapper2(Q_i , values)

// Q_i : the ID of a subquery

// values: a set E of the parts (bnv, nbnv) of the total embeddings of Q_i

// and a set V of pairs (n_k, v), where v is a candidate value for bnv[k]

begin

```

foreach embedding  $e = (bnv, nbnv)$  in  $E$  do
  foreach instance  $bnv'$  of  $bnv$  using the values in  $V$  do
    emit ( $[bnv', (Q_i, nbnv)]$ );
end.

```

Example 5.2.4. (Continued from Example 5.2.3). The mapper that works for the subquery Q_1 (i.e. the key is Q_1), gets a list of values that contain the embeddings of Q_1 in G :

```

(<Person1,*,Journal1>, <Article1,*,Title1,*>)
(<Person2,*,Journal1>, <Article1,*,Title1,*>)
(<Person4,*,Journal1>, <Article1,*,Title1,*>)

```

and the values of missing border nodes emitted by the mappers and reducers of the Phase 1 directly to the mappers of Phase 2 (see Example 5.2.2):

```

(n2, Article1), (n2, Article2), (n2, Article3)

```

The **mapper2** emits the following (key,value) pairs to the **reducer2** (reducer of Phase 2) by completing the missing node values:

```

key = (<Person1,Article1,Journal1>), value = (Q1, <Article1,*,Title1,*>)
key = (<Person1,Article2,Journal1>), value = (Q1, <Article1,*,Title1,*>)
key = (<Person1,Article3,Journal1>), value = (Q1, <Article1,*,Title1,*>)
key = (<Person2,Article1,Journal1>), value = (Q1, <Article1,*,Title1,*>)
key = (<Person2,Article2,Journal1>), value = (Q1, <Article1,*,Title1,*>)
key = (<Person2,Article3,Journal1>), value = (Q1, <Article1,*,Title1,*>)
key = (<Person4,Article1,Journal1>), value = (Q1, <Article1,*,Title1,*>)
key = (<Person4,Article2,Journal1>), value = (Q1, <Article1,*,Title1,*>)
key = (<Person4,Article3,Journal1>), value = (Q1, <Article1,*,Title1,*>)

```

The mapper that works for the subquery Q_2 (i.e. the key is Q_2), receives a list of values containing the following embeddings of Q_2 in G :

```

(<Person4,Article3,Journal2>, <*,*,*, "2008">)
(<Person2,Article2,Journal1>, <*,*,*, "2008">)
(<Person3,Article2,Journal1>, <*,*,*, "2008">)

```

Notice that Q_2 has no missing border nodes. The mapper emits the following (key,value) pairs to the reducers of Phase 2:

```

key = (<Person4,Article3,Journal2>), value = (Q2, <*,*,*, "2008">)
key = (<Person2,Article2,Journal1>), value = (Q2, <*,*,*, "2008">)
key = (<Person3,Article2,Journal1>), value = (Q2, <*,*,*, "2008">)

```

The mapper that works for the subquery Q_3 (i.e. the key is Q_3), receives a list of values that contain the embeddings of Q_3 in G :

```

(<Person4,Article1,*>, <*,Person1,*,*>)

```

$(\langle \text{Person2}, \text{Article2}, * \rangle, \langle *, \text{Person3}, *, * \rangle)$

and the values of missing border nodes:

$(n3, \text{Journal1}), (n3, \text{Journal2})$

The **mapper2** emits the following $(\text{key}, \text{value})$ pairs to the **reducer2** (reducer of Phase 2) by completing the missing node values:

$\text{key} = (\langle \text{Person4}, \text{Article1}, \text{Journal1} \rangle), \text{value} = (Q3, \langle *, \text{Person1}, *, * \rangle)$

$\text{key} = (\langle \text{Person4}, \text{Article1}, \text{Journal2} \rangle), \text{value} = (Q3, \langle *, \text{Person1}, *, * \rangle)$

$\text{key} = (\langle \text{Person2}, \text{Article2}, \text{Journal1} \rangle), \text{value} = (Q3, \langle *, \text{Person3}, *, * \rangle)$

$\text{key} = (\langle \text{Person2}, \text{Article2}, \text{Journal2} \rangle), \text{value} = (Q3, \langle *, \text{Person3}, *, * \rangle)$

Concerning the reducers of Phase 2, each reducer gets as input embeddings for each sub-query that are compatible (each one of them assigns the values in the key of the reducer to the border nodes of the query). The embeddings (one for each subquery in $(Q_1, \dots, Q_n)$) are joined to construct the final answers of Q :

reducer2(key, values)

// key: a tuple of border node values

// values: pairs of the form $(Q_i, \text{partial embedding for non-border nodes})$

begin

foreach join obtained by using one embedding for each subquery **do**

 Emit the result produced by this join;

end.

Example 5.2.5. (Continued from Example 5.2.4).

The reducer with key $\langle \text{Person2}, \text{Article2}, \text{Journal1} \rangle$ receives the following list of values:

$[(Q1, \langle \text{Article1}, *, \text{Title1}, * \rangle),$

$(Q2, \langle *, *, *, "2008" \rangle),$

$(Q3, \langle *, \text{Person3}, *, * \rangle)]$

and constructs the unique embedding of Q in G :

$(\langle \text{Person2}, \text{Article2}, \text{Journal1} \rangle, \langle \text{Article1}, \text{Person3}, \text{Title1}, "2008" \rangle)$

The remaining 11 reducers do not return any answer (they don't receive values for at least one subquery).

5.3 Discussion

Due to the specific form in which the user query Q is decomposed, namely the generalized star queries, the eval-STARS algorithm can compute embeddings

more efficiently than QEJPE-algorithm computes partial embeddings of the subqueries of Q .

Notice also that ***mapper1*** computes and emits directly to ***mapper2*** total embeddings of the subqueries that map their central nodes to non-border nodes of the data graph segment. This is also an advantage of the eval-STARS algorithm compared with the QEJPE-algorithm.

Chapter 6

Query Evaluation by Data Decomposition Using Replication

In this Chapter we present a query evaluation algorithm, called QE-with-Redundancy Algorithm, based on the idea of employing a specific form of data decomposition which allows replication (in a controlled form) which facilitates the efficient evaluation of (a specific form) of queries.

6.1 The QE-with-Redundancy Algorithm

In this Section we propose a query evaluation approach, called *QE-with-Redundancy*, which uses a specific form of replication in the data graph decomposition to efficiently answer queries. More specifically:

- (a) Data graphs are decomposed into data graph segments in which replication of the data triples is allowed. Data triples are replicated in such a way that all the answers to a special form of queries, namely *subject-object star queries*, can be obtained from a single data segment. The partition of the data graph is specified by an arbitrary partition of the data nodes, while data segments consist of the in- and out-edges of each block of nodes. Therefore, triples containing nodes that are in two different blocks occur in both segments of the data graph corresponding to these blocks. This

redundancy, as we show, ensures that the subject-object star subqueries can be easily evaluated over each segment, independently.

- (b) Each query posed by the user is decomposed into a tuple of subject-object star subqueries.

In the evaluation strategy presented in this Section, our aim is to construct the embeddings of a query Q in a data graph G , by appropriately combining embeddings (i.e. joining compatible embeddings) of so-subqueries of Q over the proper subgraphs of G .

The following lemma refers to the compatibility of embeddings:

Lemma 6.1.1. *Let $\mathcal{D}_Q = (Q_1, \dots, Q_n)$, with $n \geq 1$, be a query decomposition of a query graph Q and $\mathcal{D}_G = (G_1, \dots, G_m)$, with $m \geq 1$, be a data graph decomposition of a data graph G . Let e_{Q_i} and e_{Q_j} be two embeddings of the subqueries Q_i and Q_j respectively, with $1 \leq i \neq j \leq n$, on two (not necessarily different) graph segments D_k and D_l in $\mathcal{D}_G$. Let $\mathcal{B}(Q_i, Q_j)$ be the border nodes of Q_i , Q_j . Then e_{Q_i} and e_{Q_j} are compatible if and only if for each node $v \in \mathcal{B}(Q_i, Q_j)$, it holds that $e_{Q_i}(v) = e_{Q_j}(v)$.*

Proof. It immediately follows from Definitions 3.4.2 and 3.3.2. $\square$

In the following definition we present a decomposition scheme for a data graph G , called *star-oriented decomposition* (or simply *s-decomposition*).

Definition 6.1.1. *A star-oriented decomposition (or s-decomposition for short) of a data graph G is a tuple of graphs $\mathcal{D}_G = (G_1, \dots, G_m)$, with $m \geq 1$, if $\mathcal{N}_{\mathcal{P}} = (N_1, \dots, N_m)$ is a partition of the nodes in $\mathcal{N}(G) - L$ and for each i , with $1 \leq i \leq m$, $G_i = \{t \mid t = (s, p, o) \text{ and } t \in G \text{ and } s \in N_i \text{ or } o \in N_i\}$. Subgraphs $G_1, \dots, G_m$ are called s-graph segments. A node in $\mathcal{N}(G_i) - L - N_i$ is called a replicated node in G_i . A replicated triple $t = (s, p, o)$ in a s-graph segment G_i is a data triple in G_i such that either s or o is a replicated node.*

In the following, the set of replicated nodes in a s-graph segment G_i is denoted by $\mathcal{R}_N(G_i)$. The replicated nodes of a data graph G is $\mathcal{R}_N(G) = \bigcup_i \mathcal{R}_N(G_i)$. Similarly, the set of replicated triples in a s-graph segment G_i is denoted by $\mathcal{R}_t(G_i)$. Finally, replicated triples of a data graph G is $\mathcal{R}_t(G) = \bigcup_i \mathcal{R}_t(G_i)$.

Example 6.1.1. *Fig. 6.1 shows an s-decomposition of the data graph G of Fig. 3.1(a), which is based on the following partition of the set of nodes in $\mathcal{N}(G) - L$:*

$$\begin{aligned}
N_1 &= \{Article1, Article3, Journal2, Person4\} \\
N_2 &= \{Person1, Person2, Person3\} \\
N_3 &= \{Article2, Journal1\}.
\end{aligned}$$

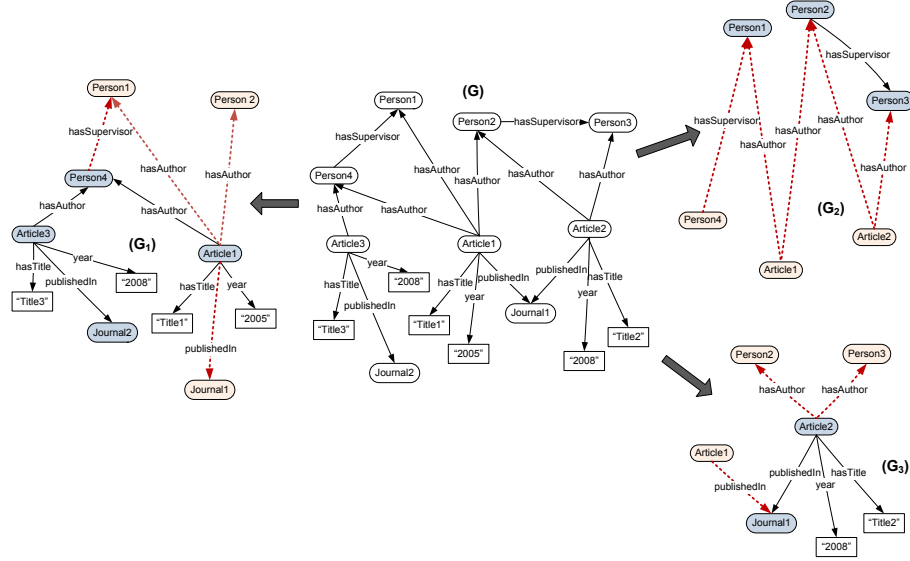


Figure 6.1: An s-decomposition of the data graph G of Fig. 3.1.

The grey colored nodes in the segments G_1 , G_2 , and G_3 correspond to the nodes in N_1 , N_2 and N_3 , respectively, while the pink colored nodes are the replicated nodes. Finally, the dashed lines in the graph segments correspond to replicated data triples.

Consider now the query graph Q appearing in the right part of Fig. 3.1. It is easy to see that we cannot obtain the solution described in Example 3.1.1 by finding an embedding of Q in a single graph segment of G appearing in Fig. 6.1 (as such an embedding does not exist).

The following lemma presents some interesting properties of the star-oriented decomposition of a data graph.

Lemma 6.1.2. *Let $\mathcal{D}_G = (G_1, \dots, G_m)$, with $m \geq 1$, be an s-decomposition of a data graph G based on the partition $\mathcal{N}_P = (N_1, \dots, N_m)$ of the nodes in $\mathcal{N}(G) - L$. Then the following hold:*

1. $(\mathcal{N}(G_i) - L) \supseteq N_i$, for each i , with $1 \leq i \leq m$.

2. $\bigcup_{i \leq m} \mathcal{N}(G_i) = \mathcal{N}(G)$
3. $\bigcup_{i \leq m} G_i = G$
4. Consider a node $s \in \mathcal{R}_N(G_i)$. Then for each triple $t = (s, p, o) \in G_i$ it holds that $o \in N_i$ and $t \in \mathcal{R}_t(G_i)$.
5. Consider a node $o \in \mathcal{R}_N(G_i)$. Then for each triple $t = (s, p, o) \in G_i$ it holds that $s \in N_i$ and $t \in \mathcal{R}_t(G_i)$.
6. For each node $v \in \mathcal{R}_N(G_i)$, with $1 \leq i \leq m$ there exists an index j , with $1 \leq j \leq m$ and $i \neq j$, such that $v \in N_j$.
7. For each triple $t \in \mathcal{R}_t(G_i)$, with $1 \leq i \leq m$, there exists an index j , with $1 \leq j \leq m$ and $i \neq j$, such that $t \in G_j$.

Proof.

Proof of 1: It immediately follows from Definition 6.1.1.

Proof of 2: Let v be a node in $\bigcup_{i \leq m} \mathcal{N}(G_i)$. Then, there exist a triple $(s, p, o) \in G_i$, for some i , such that $v = s$ or $v = o$. Since $G_i \subseteq G$, we have $(s, p, o) \in G$, which implies that $s, o \in \mathcal{N}(G)$. Therefore, $v \in \mathcal{N}(G)$.

Now let v a node in $\mathcal{N}(G)$. Then, there exist a triple $(s, p, o) \in G$, such that $v = s$ or $v = o$. Since s is the subject of this triple, it must be $s \in \mathcal{N}(G) - L$, which implies that $s \in N_i$, for some i . Therefore, $(s, p, o) \in G_i$ and thus $s, o \in \mathcal{N}(G_i)$. Consequently, $v \in \bigcup_{i \leq m} \mathcal{N}(G_i)$.

Proof of 3: From Definition 6.1.1 we conclude that $\bigcup_{i \leq m} G_i \subseteq G$. To prove the inverse let $t = (s, p, o)$ be a triple in G . Then $s \in (\mathcal{N}(G) - L)$. Thus $s \in N_i$ for some i with $1 \leq i \leq n$. Hence, by construction of the s-segments, $t \in G_i$ and therefore $t \in \bigcup_{i \leq m} G_i$. Therefore, $G \subseteq \bigcup_{i \leq m} G_i$.

Proof of 4: It immediately follows from Definition 6.1.1.

Proof of 5: It immediately follows from Definition 6.1.1.

Proof of 6: As $v \in \mathcal{R}_N(G_i)$, from Definition 6.1.1 we conclude that $v \notin N_i$. But as v is a node in $\mathcal{N}(G) - L$, the node v should belong to another set N_j , with $j \neq i$, of the partition of the nodes in $\mathcal{N}(G) - L$.

Proof of 7: Assume that t is of the form $t = (s, p, o)$. As $t \in \mathcal{R}_t(G_i)$, from Definition 6.1.1 we see that either s or o is a replicated node in $\mathcal{R}_N(G_i)$. Assume that the replicated node is s . Then, from (6) we conclude that there is an index $j \neq i$ such that $s \in N_j$. Then, from Definition 6.1.1, we conclude that $t \in G_j$. In a similar way we reach the same conclusion by assuming that o is the replicated node. $\square$

The following theorem relates the embeddings of the so-queries obtained from graph segments to the embeddings of the query Q on the graph G .

Theorem 6.1.1. *Let $\mathcal{D}_Q = (Q_1, \dots, Q_n)$, with $n \geq 1$, be a query decomposition of a query graph Q , such that each Q_i , with $1 \leq i \leq n$, is an so-query. Let also $\mathcal{D}_G = (G_1, \dots, G_m)$, with $m \geq 1$, be an s -decomposition of a data graph G . Then e is a total embedding of Q in G if and only if e is the join of $e_1, \dots, e_n$, where $e_1, \dots, e_n$ are mutually compatible embeddings such that for each i , with $1 \leq i \leq n$, e_i is a total embedding of Q_i in some segment G_j , where $1 \leq j \leq m$.*

Proof. For the one direction, assume that e is a (total) embedding of Q in G . For every i , with $1 \leq i \leq n$, define e_i to be the restriction of e in Q_i (that is, $e_i : \mathcal{N}(Q_i) \rightarrow \mathcal{N}(G)$, with $e_i(v) = e(v)$ for every node $v \in \mathcal{N}(Q_i)$). Obviously e_i is a total mapping. Furthermore, for every non-variable element $v \in \mathcal{N}(Q_i)$ it is $e_i(v) = e(v) = v$ and for every triple $(v_1, p, v_2) \in Q_i$ it is $(e_i(v_1), p, e_i(v_2)) = (e(v_1), p, e(v_2)) \in G$, which implies that e_i is actually an embedding of Q_i in G .

As Q_i is an so-query, let $C(Q_i)$ be the central node of Q_i and $e(C(Q_i)) \in \mathcal{N}(G)$ the image of $C(Q_i)$ in G . From Definition 6.1.1 we conclude that $e(C(Q_i)) \in N_j$, for some N_j with $1 \leq j \leq m$ and that e_i is an embedding of Q_i in G_j .

We next prove that the embeddings e_i , with $1 \leq i \leq n$ are mutually compatible and their join is e . By construction, for every i, j with $i \neq j$, if $v \in \mathcal{N}(Q_i) \cap \mathcal{N}(Q_j)$ then it is $e_i(v) = e_j(v) = e(v)$, which implies that e_i and e_j are compatible. Therefore, the join e' of $e_1, \dots, e_n$ exists. It remains to show that $e' = e$. Consider an arbitrary $v \in \mathcal{N}(Q)$. Then v appears in some triple $t \in Q$. Since $\mathcal{D}_Q$ is a decomposition of Q , there exists some i such that $t \in Q_i$. Thus, $v \in \mathcal{N}(Q_i)$, which implies that $e_i(v)$ is defined. From the definition of join, it follows that $e'(v) = e_i(v)$, which implies $e'(v) = e(v)$.

For the other direction, assume that for each i , with $1 \leq i \leq n$, there is an embedding e_i for the subquery Q_i in some graph segment G_j . Assume also that $e_1, \dots, e_n$ are mutually compatible embeddings and let e be their join. We will prove that e is an embedding of Q in G . It is easy to see that for every $v \in \mathcal{N}(Q)$ there exists some i such that $e_i(v)$ is defined, which implies that $e(v)$ is also defined. Thus, e is a total mapping.

We next show that e is an embedding of Q in G . Consider any non-variable element $v \in \mathcal{N}(G)$. Since e is total, $e(v)$ is defined. From the definition of join, there must be some i such that $e(v) = e_i(v)$, which implies $e(v) = v$ (since e_i is an embedding).

Finally, let (v_1, p, v_2) be a triple in Q . Since $\mathcal{D}_Q$ is a decomposition of Q , (v_1, p, v_2) belongs to some Q_i . Since e_i is a total embedding of Q_i in an s-segment of G , it is also a total embedding of Q_i in G . Thus, $(e_i(v_1), p, e_i(v_2)) \in G$, which implies $(e(v_1), p, e(v_2)) \in G$. $\square$

The above theorem suggests the following strategy for the evaluation of a query Q on a data graph G , called *QE-with-Redundancy*. *QE-with-Redundancy* strategy assumes a star-oriented decomposition of the data graph G . To obtain such a decomposition we assume an arbitrary partition $\mathcal{N}_{\mathcal{P}} = (N_1, \dots, N_m)$, with $m \geq 1$ of the nodes in $\mathcal{N}(G) - L$. Then we decompose the data graph G into a tuple of graph segments $\mathcal{D}_G = (G_1, \dots, G_m)$, such that $\mathcal{D}_G$ is a star-oriented decomposition of G (as defined in Definition 6.1.1).

The *QE-with-Redundancy* strategy consists of the following steps:

- Step 1:** Decompose the query Q into a tuple of queries $\mathcal{D}_Q = (Q_1, \dots, Q_n)$, with $n \geq 1$, such that each query in $\mathcal{D}_Q$ is a subject-object star query.
- Step 2:** Compute all possible embeddings of each subquery in $\mathcal{D}_Q$ on every segment in $\mathcal{D}_G$.
- Step 3:** Compute the embeddings of Q on G by joining compatible embeddings of the subqueries $Q_1, \dots, Q_n$.

It is important to note that the algorithm is independent of the choice of the specific partition $\mathcal{N}_{\mathcal{P}}$ of the nodes in $\mathcal{N}(G) - L$, used for the data graph decomposition, as well as of the specific query decomposition strategy (employed in Step 1).

6.2 A Map-Reduce Implementation of the QE-with-Redundancy Algorithm

In this Section, we present an implementation of the QE-with-Redundancy query evaluation algorithm, presented in Section 6.1, based on the MapReduce programming framework. Recall that, for the implementation of the algorithm we assume a star-oriented decomposition (s-decomposition) of the data graph G and a (possibly redundant) decomposition of the query Q posed by the user into a set of *subject-object star subqueries* $\{Q_1, \dots, Q_n\}$, with $n \geq 1$. The implementation of the algorithm consists of a preprocessing phase followed by one and a half

Map-Reduce phase. The first phase of our algorithm takes advantage of the star form of the subqueries and focuses on evaluating the star subqueries over the input segments. The results of the sub-queries are emitted to the second phase, which combines them properly in order to produce the answers of the initial query.

6.2.1 The preprocessing phase

In the preprocessing phase the users' query Q is decomposed into a set of so-queries $\{Q_1, \dots, Q_n\}$, with $n \geq 1$, and the auxiliary structures presented in Subsection 4.2.1 are constructed. Preprocessing phase emits the above to the mappers of Phase 1 with key the pair $(subqueryID, SegmentID)$.

Example 6.2.1. *To present the QE-with-redundancy algorithm, we will use again the query Q and its decomposition into three so-queries presented in Fig. 5.1. The query prototypes, the MBN list and the tuple of common border nodes of appearing in these subqueries are the same as in Example 5.2.1.*

Concerning the data graph decomposition, to present the algorithm we will use the data graph segments (s-segments) obtained by decomposing the data graph G as presented in Fig. 6.1.

6.2.2 Phase 1 of the algorithm

The first phase of the algorithm computes the embeddings of the so-queries $Q_1, \dots, Q_n$ in G locally in each star graph segment G_j of G .

Each *mapper* in phase 1 gets as input an s-graph segment G_j , an so-query Q_i , the MBN list, and the tuple $\mathcal{CB}(Q)$ and computes the embeddings of Q_i into G_j . The embeddings computed are directly emitted to the mappers of Phase 2. Similarly, the values of the nodes in MBN are emitted to the mappers of Phase 2. Notice that the instances of the nodes in $\mathcal{CB}(Q)$ take part in the keys of the (key, value) pairs emitted to the mappers of Phase 2.

```
mapper1(( $Q_i, G_j$ ), ( $G_jData, SubqueryInfo, MBN, \mathcal{CB}(Q)$ ))
//( $Q_i, G_j$ ):  $Q_i/G_j$  is the ID of a subquery/data segment
//  $G_jData$ : the content of the data graph segment  $G_j$ 
//  $SubqueryInfo$ : prototypes of the subqueries of  $Q$ 
//  $MBN$ : the list of missing border nodes
//  $\mathcal{CB}(Q)$  is the tuple of common border nodes of  $Q$ 
begin
```

```

compute  $E = \{e \mid e \text{ is an embedding of } Q_i \text{ in } G_j\text{Data}\}$ 
for each embedding  $e = (bnv, nbnv)$  in  $E$  do
  begin
    if ( $MBN! = []$ ) then
      emitToMapper2( $[(Q_i, e(\mathcal{CB}(Q))), (bnv, nbnv)]$ );
      for  $k = 1$  to  $|bnv|$  do
        if ( $bnv[k] \neq ''$ ) then
          foreach  $(n_k, Q_j)$  in  $MBN$  do
            emitToMapper2( $[(Q_j, e(\mathcal{CB}(Q))), (n_k, bnv[k])]$ );
        else
          emitToReducer2( $[bnv, (Q_i, nbnv)]$ );
        end
      end
    end.

```

Example 6.2.2. (Continued from Example 6.2.1). This example shows the results obtained by the application of **mapper1** on the pairs (Q_i, G_j) , where Q_i is an so-query and G_j is a graph segment.

The following three embeddings of Q_1 into G_1 are computed by the algorithm:

$e_1 = (<Person1, *, Journal1>, <Article1, *, Title1, *>)$

$e_2 = (<Person2, *, Journal1>, <Article1, *, Title1, *>)$

$e_3 = (<Person4, *, Journal1>, <Article1, *, Title1, *>)$

For e_1 the algorithm emits the following (key, value) pair to **mapper2**:

key = $(Q, Person1)$, value = $(<Person1, *, Journal1>, <Article1, *, Title1, *>)$

Besides, based on the MBN list and the $\mathcal{CB}(Q)$, which, in the preprocessing phase have been computed to $MBN = [(n_2, Q_1), (n_3, Q_3)]$ and $\mathcal{CB}(Q) = \{n_1\}$,

mapper1 also emits to **mapper2** the following key value pair:

key = $(Q3, Person1)$, value = $(n3, Journal1)$

Similarly, **mapper1** also emits the following (key, value) pairs based on the embeddings e_2 and e_3 :

key = $(Q1, Person2)$, value = $(<Person2, *, Journal1>, <Article1, *, Title1, *>)$

key = $(Q3, Person2)$, value = $(n3, Journal1)$

key = $(Q1, Person4)$, value = $(<Person4, *, Journal1>, <Article1, *, Title1, *>)$

key = $(Q3, Person4)$, value = $(n3, Journal1)$

Concerning query Q_1 there are no embeddings in segments G_2 and G_3 . Thus nothing is emitted by the corresponding mappers.

The following embedding of Q_2 into G_1 is computed (among others) by the algorithm:

$e'1 = (<Person4, Article3, Journal2>, <*, *, *, "2008">)$

For e'_1 the algorithm emits the following(key,value) pair to **mapper2**:

key = (Q2, Person4), value = (<Person4, Article3, Journal2>, <*, *, *, "2008">)

Besides, based on the MBN list and the $CB(Q)$, **mapper1** also emits to **mapper2** the following key value pair:

key = (Q1, Person4), value = (n2, Article3)

key = (Q3, Person4), value = (n3, Journal2)

Query Q_2 has no embeddings in segment G_2 ; hence nothing is emitted in this case.

The following two embedding of Q_2 into G_3 are computed by the algorithm:

$e'2 = (<Person2, Article2, Journal1>, <*, *, *, "2008">)$

$e'3 = (<Person3, Article2, Journal1>, <*, *, *, "2008">)$

As above, based on these embeddings as well as on the content of the MBL list and the $CB(Q)$, **mapper1** also emits to **mapper2** the following key value pairs:

key = (Q2, Person2), value = (<Person2, Article2, Journal1>, <*, *, *, "2008">)

key = (Q1, Person2), value = (n2, Article2)

key = (Q3, Person2), value = (n3, Journal1)

key = (Q2, Person3), value = (<Person3, Article2, Journal1>, <*, *, *, "2008">)

key = (Q1, Person3), value = (n2, Article2)

key = (Q3, Person3), value = (n3, Journal1)

The following embedding of Q_3 into G_1 is computed by the algorithm:

$e''1 = (<Person4, Article1, *>, <*, Person1, *, *>)$

For e''_1 the algorithm emits the following(key,value) pair to **mapper2**:

key = (Q3, Person4), value = (<Person4, Article1, *>, <*, Person1, *, *>)

Besides, based on the MBN list and the $CB(Q)$, **mapper1** also emits to **mapper2** the following key value pair:

key = (Q1, Person4), value = (n2, Article1)

The following embeddings of Q_3 into G_2 are computed by the algorithm:

$e''1 = (<Person4, Article1, *>, <*, Person1, *, *>)$

$e''2 = (<Person2, Article2, *>, <*, Person3, *, *>)$

As above, based on these embeddings as well as on the content of the MBL list and the $CB(Q)$, **mapper1** also emits to **mapper2** the following key value pairs:

key = (Q3, Person4), value = (<Person4, Article1, *>, <*, Person1, *, *>)

key = (Q1, Person4), value = (n2, Article1)

key = (Q3, Person2), value = (<Person2, Article2, *>, <*, Person3, *, *>)

key = (Q1, Person2), value = (n2, Article2)

Query Q_3 has no embeddings in segment G_3 ; hence nothing is emitted by this mapper.

6.2.3 Phase 2 of the algorithm

Phase 2 of the algorithm is similar to the Phase 2 of the eval-STARS algorithm.

Each *mapper* in Phase 2 gets as input all the embeddings of a specific subquery Q_i which have the same values for the nodes in $\mathcal{CB}(Q)$; moreover for each border node that does not occur in Q_i it gets as input the values assigned to this node by the embeddings of the other subqueries. It fills in their missing border node values using the corresponding values in the input, and emits the resulted embeddings to the reducers of Phase 2. The key is the tuple of the border node values, which implies that two embeddings are emitted to the same reducer if and only if they are compatible.

```

mapper2(( $Q_i, e(\mathcal{CB}(Q))$ ), values)
//  $Q_i$ : the ID of a subquery
// values: a set  $E$  of the parts ( $bnv, nbnv$ ) of the embeddings
//         of  $Q_i, e(\mathcal{CB}(Q))$  and a set  $V$  of pairs  $(n_k, v)$ ,
//         where  $v$  is a candidate value for  $bnv[k]$ 
begin
  foreach embedding  $e = (bnv, nbnv)$  in  $E$  do
    foreach ground instance  $bnv'$  of  $bnv$  using the values in  $V$  do
      emit([ $bnv', (Q_i, nbnv)$ ]);
end.

```

Example 6.2.3. (Continued from Example 6.2.2). This example shows the application of **mapper2**.

The mapper applied for the key $(Q_1, Person1)$ gets the value:

$(\langle Person1, *, Journal1 \rangle, \langle Article1, *, Title1, * \rangle)$

but it does not get any value for the missing border nodes. As $V = \emptyset$ no ground instances of $(\langle Person1, *, Journal1 \rangle)$ can be found. Therefore this mapper does not emit (key, value) pairs to **reducer2**.

The mapper applied for the key $(Q_1, Person2)$ gets the value:

$(\langle Person2, *, Journal1 \rangle, \langle Article1, *, Title1, * \rangle)$
 $(n2, Article2)$

This mapper constructs the instance $\langle Person2, Article2, Journal1 \rangle$ of bnv and emits the following (key, value) pair to **reducer2**:

key = $\langle Person2, Article2, Journal1 \rangle$, value = $(Q1, \langle Article1, *, Title1, * \rangle)$

The mapper applied for the key $(Q_1, Person3)$ gets the values:

$(n2, Article2)$

As $E = \emptyset$ this mapper emits nothing to **reducer2**.

The mapper applied for the key $(Q_1, \text{Person4})$ gets (after eliminating duplicates) the values:

$(\langle \text{Person4}, *, \text{Journal1} \rangle, \langle \text{Article1}, *, \text{Title1}, * \rangle)$
 $(n2, \text{Article3})$
 $(n2, \text{Article1})$

This mapper emits the following (key, value) pairs to **reducer2**:

key = $\langle \text{Person4}, \text{Article3}, \text{Journal1} \rangle$, value = $(Q1, \langle \text{Article1}, *, \text{Title1}, * \rangle)$
key = $\langle \text{Person4}, \text{Article1}, \text{Journal1} \rangle$, value = $(Q1, \langle \text{Article1}, *, \text{Title1}, * \rangle)$

The mapper applied for the key $(Q_2, \text{Person2})$ gets the value:

$(\langle \text{Person2}, \text{Article2}, \text{Journal1} \rangle, \langle *, *, *, "2008" \rangle)$

This mapper emits the following (key, value) pair to **reducer2**:

key = $\langle \text{Person2}, \text{Article2}, \text{Journal1} \rangle$, value = $(Q2, \langle *, *, *, "2008" \rangle)$

The mapper applied for the key $(Q_2, \text{Person3})$ gets the value:

$(\langle \text{Person3}, \text{Article2}, \text{Journal1} \rangle, \langle *, *, *, "2008" \rangle)$

This mapper emits the following (key, value) pair to **reducer2**:

key = $\langle \text{Person3}, \text{Article2}, \text{Journal1} \rangle$, value = $(Q2, \langle *, *, *, "2008" \rangle)$

The mapper applied for the key $(Q_2, \text{Person4})$ gets the values:

$(Q2, \text{Person4})$
 $(\langle \text{Person4}, \text{Article3}, \text{Journal2} \rangle, \langle *, *, *, "2008" \rangle)$

This mapper emits the following (key, value) pair to **reducer2**:

key = $\langle \text{Person4}, \text{Article3}, \text{Journal2} \rangle$, value = $(Q2, \langle *, *, *, "2008" \rangle)$

The mapper applied for the key $(Q_3, \text{Person1})$ gets the value:

$(n3, \text{Journal1})$

As $E = \emptyset$ this mapper emits nothing to **reducer2**.

The mapper applied for the key $(Q_3, \text{Person2})$ gets the value:

$(\langle \text{Person2}, \text{Article2}, * \rangle, \langle *, \text{Person3}, *, * \rangle)$
 $(n3, \text{Journal1})$

This mapper emits the following (key, value) pair to **reducer2**:

key = $\langle \text{Person2}, \text{Article2}, \text{Journal1} \rangle$, value = $(Q3, \langle *, \text{Person3}, *, * \rangle)$

The mapper applied for the key $(Q_3, \text{Person3})$ gets the value:

$(n3, \text{Journal1})$

As $E = \emptyset$ this mapper emits nothing to **reducer2**.

The mapper applied for the key $(Q_3, \text{Person4})$ gets the values:

$(\langle \text{Person4}, \text{Article1}, * \rangle, \langle *, \text{Person1}, *, * \rangle)$
 $(n3, \text{Journal1})$
 $(n3, \text{Journal2})$

This mapper emits the following (key, value) pairs to **reducer2**:

```
key = <Person4, Article1, Journal1>, value = (Q3, <*, Person1, *, *>)
key = <Person4, Article1, Journal2>, value = (Q3, <*, Person1, *, *>)
```

Concerning the *Reducer of Phase 2*, each reducer gets as input embeddings for each subquery that are compatible as the key for the reducer is a tuple of values for all border nodes of the query Q . The embeddings (one for each subquery $Q_1, \dots, Q_n$) are joined to construct the final answers of Q :

```
reducer2(key, values)
// key: a tuple of values for the border nodes of Q
// values: pairs of the form (Qi, partial embedding for non-border nodes)
begin
  foreach join obtained by using one embedding for each subquery do
    Emit the result produced by this join;
end.
```

Example 6.2.4. (Continued from Example 6.2.3). The reducer with key:

$< \text{Person2}, \text{Article2}, \text{Journal1} >$ receives the list:

```
[(Q1, <Article1, *, Title1, *>), (Q2, <*, *, *, "2008">), (Q3, <*, Person3, *, *>)]
```

As this lists contains an embedding for each subquery, we join them and obtain the following embedding of the query Q :

```
(<Person2, Article2, Journal1>, <Article1, Person3, Title1, "2008">)
```

This embedding corresponds to the answer:

```
(?P1, ?A, ?J, ?P2, ?T) = (Person2, Article2, Journal1, Person3, Title1 )
```

Note that, the remaining reducers do not return any answer (they don't receive values for at least one subquery).

6.3 Query Decomposition Algorithms

In this Section, we present and analyze algorithms for decomposing queries into a set of so-subqueries. In Section 6.1, we assumed that the queries are decomposed into a set of subqueries, but we have not typically discussed any algorithm for finding such a decomposition, so far. Although the algorithms presented in Chapters 4, 5, and 6 can be used to evaluate a query over a single machine, they are designed to be efficiently applied on a distributed environment, as we will see in the next Sections. In this context, we focus on decomposition algorithms that can boost parallelization. Furthermore, although the QEJPE algorithm (Chapter 4) is quite generic and can support every query

decomposition, the algorithms presented in this Section aim to take advantage of the special, so-queries decomposition, which can be used in the evaluation algorithms eval-STARS (Chapter 5) and QE-with-Redundancy (Chapter 6).

Intuitively, the decomposition approach followed can affect the efficiency of the overall query evaluation process, since an appropriate algorithm can significantly reduce the amount of the data transferred through the network (i.e., the amount of intermediate results). For example, in the extreme scenario that we decompose the query in Figure 3.2 so that each edge defines a different subquery, it is easy to see that all the 6 edges with predicate “*hasAuthor*” are mapped by the edge-subquery $\{(?A, \text{hasAuthor}, ?W)\}$; hence, 6 embeddings are resulted by Step 2 of the QE-with-Redundancy algorithm and passed to Step 3. If, however, we decompose the query Q in such a way that at least one constant (i.e., non-variable node) is included in each subquery, the number of embeddings found in Step 2 and used in Step 3 is significantly reduced; e.g., consider the query $\{(?A, \text{hasAuthor}, ?W), (?A, \text{year}, 2008)\}$, or Q itself. Practically, the more the number of constants each query has, the less the embeddings that are found, since the constants filter out useless embeddings (i.e., partial embeddings that surely cannot be used in Step 3 to construct a total embedding). The Steps 3 and 4 of the eval-STARS algorithm operate similarly. The following proposition proves this statement.

Proposition 6.3.1. *Let Q_1 and Q_2 be two generalized star queries, such that $Q_2 \subseteq Q_1$ and each triple in the set $(Q_1 - Q_2)$ is either of the form (C, p, c) or of the form (c, p, C) , where $C = C(Q_1) = C(Q_2)$, p is a predicate, and c is not a variable. Then, for every data graph G the set of answers of Q_1 over G is a subset of the set of answers of Q_2 over G , and $n_1^e \leq n_2^e$, where n_i^e is the number of embeddings of Q_i over G , with $i = 1, 2$.*

The proof of the previous proposition is straightforwardly given by expressing both queries as conjunctive queries and checking containment of the corresponding conjunctive queries [19, 27].

To maximize the number of constants in each subquery, one could come up with the following simple decomposition algorithm (called *naive algorithm*). Let Q be a query.

Step 1: For each node n in $\mathcal{N}(Q)$, we construct the star query Q_n such that Q_n includes all the edges in Q of either the form (n, p, m) or the form (m, p, n) , where $m \in \mathcal{N}(Q)$. Let S_Q be the set including all the subqueries constructed by this process.

Step 2: We, then, remove from S_Q the subqueries that are not so-queries.

It is easy to see that the remaining subqueries in S_Q form a decomposition $\mathcal{D}_Q$ of Q that can be used in both QE-with-Redundancy and eval-STARS algorithms.

Proposition 6.3.2. *Considering a query Q , the naive algorithm results a decomposition $\mathcal{D}_Q$ of Q such that each query in $\mathcal{D}_Q$ is an so-query.*

The proof of the Proposition 6.3.2 is straightforward since each edge of Q will be at least in the subquery centered by its subject. Furthermore, it is easy to see that the naive algorithm results a query decomposition that maximizes the number of constants in each star subquery, since each subquery is constructed by a query node along with all of its adjacent edges.

This algorithm, however, results a quite large number of subqueries as at the worst-case scenario one subquery for each query edge is obtained (the subject of each query edge, may introduce a new subquery), and does not limit the number of variables in each subquery, which might impact the overall performance of the query evaluation, as we will see in the next Sections.

In the following Sections, we study two additional parameters, the number of variables into each subquery and the number of subqueries in the decomposition, in order to provide an effective decomposition approach. In particular, in Subsection 6.3.1, we present an algorithm that decomposes a query in a way that the number of variables do not exceed a given threshold. Subsection 6.3.2 discusses multiple algorithms that aim to reduce the number of subqueries into the decomposition.

6.3.1 Subqueries with a limited number of variables

In this Section, we present a decomposition algorithm that aims to keep the number of the variables in each subquery as low as possible. As Proposition 6.3.1 shows, by decomposing into so-subqueries with large number of constants we can achieve significant improvement in the overall performance of both QE-with-Redundancy and eval-STARS algorithms. However, we might not have the same result if we choose star subqueries with large number of variables. To see which is the impact of the number of variables into the overall query evaluation process we start our analysis with an example.

Consider the simple query $Q = \{(c, p_1, ?X), (c, p_2, ?Y), (?X, p_3, ?Y)\}$ and the data graph $G = \{(c, p_1, c_{11}), (c, p_1, c_{12}), (c, p_1, c_{13}), (c, p_2, c_{21}), (c, p_2, c_{22}), (c, p_2, c_{23})\}$, where p_1, p_2, p_3 are two predicates and c, c_{ij} are either URIs or

literals. Suppose now a decomposition $\mathcal{D} = \{Q_1, Q_2\}$ of Q , such that $Q_1 = \{(c, p_1, ?X), (c, p_2, ?Y)\}$ and $Q_2 = \{(?X, p_3, ?Y)\}$. Notice that although the answers of both Q and Q_2 are empty, there are 9 total embeddings from Q_1 to G , giving 9 answers. Looking at the algorithms eval-STARS (Steps 4 and 5) and QE-with-Redundancy (Steps 3 and 4), it is worth further decomposing Q_1 into two subqueries Q_{11} , Q_{12} , one for each edge, instead of keeping Q_1 into $\mathcal{D}$. In particular, if we replace Q_1 in $\mathcal{D}$ with Q_{11} and Q_{12} , the number of embeddings found in Step 4 of eval-STARS and passed to Step 5 (resp., found in Step 3 of QE-with-Redundancy and passed to Step 4) is 6, instead of 9 in the case we use Q_1 .

In the previous example, we saw that the presence of multiple variables in a subquery might increase the number of embeddings of this subquery in both QE-with-Redundancy and eval-STARS algorithms. Especially, if we apply these algorithms into a distributed environment, as we will see in the next Sections, we might have significant impact on the performance of each algorithm, since the communication cost might be increased tremendously from the large number of embeddings transferred through the network.

In this context, we present the *min-res* decomposition algorithm, which finds a decomposition by keeping the number of variables into each subquery at most 2. One could wonder how we come up with the threshold number 2. Typically, we want to keep the number of variables in each subquery as low as possible. If we set such a threshold to one variable, we miss edges consisting of two variables, i.e., we cannot find a valid decomposition of any given query.

Min-res algorithm decomposes a query Q into a set of so-subqueries, such that each subquery has at most two variables. It also allows replication of triples that contains at most one variable, and maximizes the number of “constraints” (triples that do not increase the number of variables in the query) in each subquery containing variables. As for the subqueries that do not contain any variable, the algorithm constructs maximal subqueries without redundant constraints. The *min-res* decomposition algorithm, in detail, is given as follows.

```

min-res( $Q$ )
//  $Q$  a query.
// The min-res function returns a decomposition  $R$  of  $Q$  consisting of so-subqueries of  $Q$ 
begin
   $T_{sub-obj} = \{t \in Q \mid t = (s, p, o) \text{ and } s, o \in \mathcal{V}(Q)\};$  // subject and object are variables
   $T_{sub} = \{t \in Q \mid t = (s, p, o) \text{ and } s \in \mathcal{V}(Q) \text{ and } o \notin \mathcal{V}(Q)\};$  // only subject is variable
   $T_{obj} = \{t \in Q \mid t = (s, p, o) \text{ and } o \in \mathcal{V}(Q) \text{ and } s \notin \mathcal{V}(Q)\};$  // only object is variable
   $T_c = \{t \in Q \mid t = (s, p, o) \text{ and } s, o \notin \mathcal{V}(Q)\};$  // subject and object are nonvariables
   $R = \emptyset;$ 

```

```

foreach  $t = (s, p, o) \in T_{sub-obj}$  do // select a maximal so-query centered at subject of  $t$ 
  begin // by adding triples that do not add variables
     $Q_s = \{t\} \cup \{t' \mid t' \in T_{sub} \text{ and } t' = (s, p', c)\} \cup \{t'' \mid t'' \in T_{obj} \text{ and } t'' = (c', p'', s)\};$ 
    //  $Q_s$  is an so-query with central node  $s$ 
     $S = \{t' \mid t' \in T_{sub} \text{ and } t' = (o, p', c)\};$ 
    If  $S = \emptyset$  then  $Q_o = \emptyset$  else
       $Q_o = \{t\} \cup S \cup \{t'' \mid t'' \in T_{obj} \text{ and } t'' = (c', p'', o)\};$ 
      //  $Q_o$  is an so-query with central node  $o$ 
    If  $|Q_s| \geq |Q_o|$  then  $Q' = Q_s$  else  $Q' = Q_o$ ;
     $R = R \cup \{Q'\};$ 
  end
   $T_{sub} = T_{sub} - \{t \mid t \in Q' \text{ and } Q' \in R\};$  // Remove from  $T_{sub}$  the triples used so far
   $T_{obj} = T_{obj} - \{t \mid t \in Q' \text{ and } Q' \in R\};$  // Remove from  $T_{obj}$  the triples used so far
  while  $T_{sub} \neq \emptyset$  do //For each member of  $T_{sub}$  construct an so-query
    begin
      extract a triple  $t = (s, p, o)$  from  $T_{sub}$ ;
       $Q' = \{t\} \cup \{t' \mid t' \in T_{sub} \text{ and } t' = (s, p', c)\} \cup \{t'' \mid t'' \in T_{obj} \text{ and } t'' = (c', p'', s)\};$ 
      If  $|Q'| = 1$  then // No other triple has  $s$  as object or subject
        begin
           $S = \{t' \mid t' \in T_c \text{ and } t' = (o, p', c)\};$ 
          If  $S \neq \emptyset$  then
             $Q' = \{t\} \cup S \cup \{t'' \mid t'' \in T_c \text{ and } t'' = (c', p'', o)\};$ 
          end
           $T_{sub} = T_{sub} - \{t \mid t \in Q'\};$  // Remove from  $T_{sub}$  the triples used in  $Q'$ 
           $T_{obj} = T_{obj} - \{t \mid t \in Q'\};$  // Remove from  $T_{obj}$  the triples used in  $Q'$ 
           $R = R \cup \{Q'\};$ 
        end
      foreach  $t = (s, p, o) \in T_{obj}$  do //For each member of  $T_{obj}$  construct an so-query
        begin
           $Q' = \{t\} \cup \{t' \mid t' \in T_c \text{ and } t' = (s, p', c)\} \cup \{t'' \mid t'' \in T_c \text{ and } t'' = (c', p'', s)\};$ 
           $R = R \cup \{Q'\};$ 
        end
       $T_c = T_c - \{t \mid t \in Q' \text{ and } Q' \in R\};$ 
      while  $T_c \neq \emptyset$  do // select maximal so-query centered at subject or object
        begin
          extract a triple  $t = (s, p, o)$  from  $T_c$ ;
           $Q_s = \{t\} \cup \{t' \mid t' \in T_c \text{ and } t' = (s, p', c)\} \cup \{t'' \mid t'' \in T_c \text{ and } t'' = (c', p'', s)\};$ 
           $S = \{t' \mid t' \in T_c \text{ and } t' = (o, p', c)\};$ 
          If  $S = \emptyset$  then  $Q_o = \emptyset$  else
             $Q_o = \{t\} \cup S \cup \{t'' \mid t'' \in T_c \text{ and } t'' = (c', p'', o)\};$ 
          If  $|Q_s| \geq |Q_o|$  then  $Q' = Q_s$  else  $Q' = Q_o$ ;
           $R = R \cup \{Q'\};$ 
           $T_c = T_c - \{t \mid t \in Q'\};$  // Remove from  $T_c$  the triples used in  $Q'$ 
        end
      return  $R$ ;
    end.

```

Intuitively, the algorithm performs as follows. Let Q be a query. Initially, for each edge t of two variables in Q , it constructs an so-query Q_s having the subject s of t as central node. All the adjacent edges of s in Q such that s is their only variable are added into the subquery. In each construction step, the possibility

to get an so-query Q_o , whose central node is the object of t , is also considered, and the query with maximum number of edges between Q_s and Q_o is finally selected. It is easy to see that the subqueries constructed in this step include 2 variables. Next, the algorithm constructs the subqueries that have a single variable as central code. These subqueries have at least one edge whose subject is the central variable. Then, the remaining query triples give so-subqueries whose central node is not a variable. Each of the subqueries constructed in this step have at most a single variable which is not the central node. Notice here that the min-res algorithm constructs subqueries with two variables only if those variables are used by an edge in Q . Note also that in each of the aforementioned steps, we build a so-subquery by initially selecting an edge from a set (e.g., the set of edges having two variables). The order the edges are selected might give different decompositions. Here, we consider an arbitrary ordering of the edges included in each set. The following proposition shows that the min-res algorithm results a decomposition of a query into a set of so-queries.

Proposition 6.3.3. *Considering a query Q , the min-res algorithm results a decomposition $\mathcal{D}_Q$ of Q such that each query in $\mathcal{D}_Q$ is an so-query.*

Proof. Consider the sets $T_{sub-obj}$, T_{sub} , T_{obj} , T_c of edges as defined in the min-res algorithm. To prove that R is a decomposition of Q , we need to show that (1) each query Q_i in R is a subquery of Q , and (2) $\bigcup_{Q_i \in R} Q_i = Q$. The first condition is straightforward since each edge of Q_i is an edge in $T_{sub-obj} \cup T_{sub} \cup T_{obj} \cup T_c$, which equals Q . To show the second condition, we need to prove that each edge t of Q is included in at least one subquery in R . Since the algorithm uses all the edges in $T_{sub-obj} \cup T_{sub} \cup T_{obj} \cup T_c$ to construct the subqueries, we have that t is included in at least one subquery in R . Besides, it is easy to see that, by construction, all queries in $\mathcal{D}_Q$ are so-queries. $\square$

Example 6.3.1. *Consider the query Q depicted in Figure 5.1. Figure 6.2 illustrates a decomposition of Q resulted by the min-res algorithm. In particular, we initially select the edge t_2 and construct Q_1 . Similarly, the queries Q_2 , Q_3 and Q_4 are given by selecting the edges t_3 , t_4 and t_5 , respectively, having two variables, as well. Q_5 and Q_6 are then constructed by selecting the corresponding edges of Q . Note here that the edge t_6 is replicated to multiple subqueries, as min-res algorithm requires, since it can reduce the number of intermediate answers through the constant “2008”.*

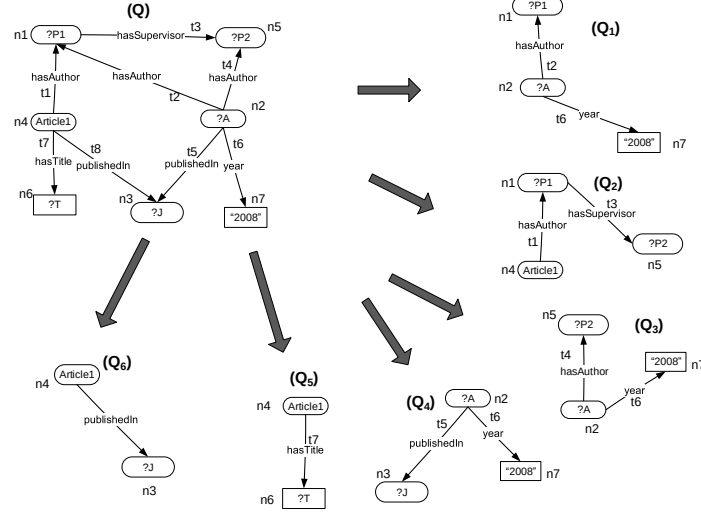


Figure 6.2: Min-res Query decomposition.

6.3.2 Reducing the number of subqueries

Unlike the min-res algorithm which minimizes the number of variables in each subquery, in this Section, we investigate algorithms that keep the number of so-subqueries as low as possible, as well as select so-subqueries with high degree. As we will see in the following, there are settings where the number of queries in the decomposition affects the overall performance of the query evaluation, since large number of subqueries might increase the amount of intermediate results. The following example presents such a case.

Example 6.3.2. Consider the query Q and the data graph G depicted in Figure 5.1 and Figure 3.2, respectively. It is easy to see that there is a single total embedding from Q to G . Suppose two decompositions $\mathcal{D}_Q^1$ and $\mathcal{D}_Q^2$ illustrated in Figure 6.2 and Figure 6.3, respectively. As we saw in Example 6.3.1, $\mathcal{D}_Q^1$ is resulted by min-res algorithm. Counting now the embeddings found for the subqueries of each decomposition over G , we have that there are 12 embeddings, in total, from queries in $\mathcal{D}_Q^1$ to G , while $\mathcal{D}_Q^2$ gives 10 embeddings. Hence, we can see that although each subquery in $\mathcal{D}_Q^1$ has minimum number of variables, the total number of embeddings is high, due to the large number of subqueries.

To construct a decomposition with minimum number of subqueries, we follow an approach based on the naive algorithm. In particular, considering a query Q ,

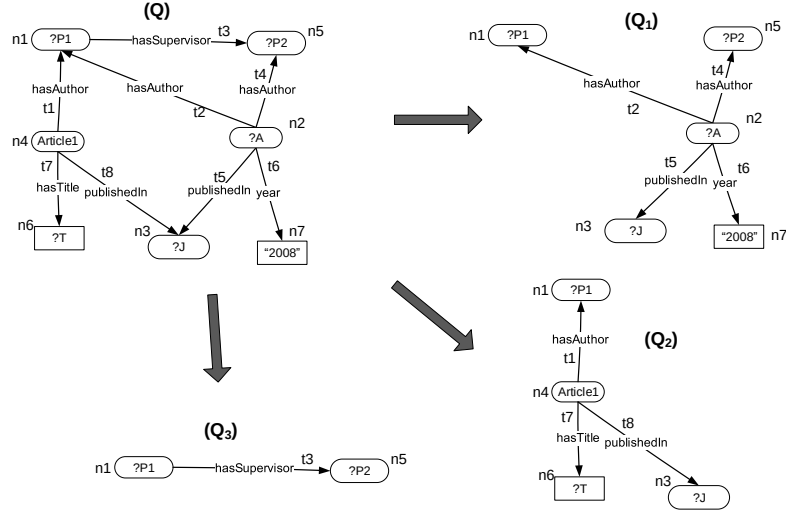


Figure 6.3: Min-subquery decomposition.

a simple algorithm, called *min-subquery* decomposition algorithm, computing a decomposition with minimum number of so-subqueries is given as follows.

Step 1: We initially apply the naive algorithm and get a decomposition $\mathcal{D}_N$.

Step 2: Then, we construct the set $\mathcal{S}$ including all the subsets $\mathcal{D}$ of $\mathcal{D}_N$ such that the queries in $\mathcal{D}$ cover all the edges of Q ; i.e., $\bigcup_{Q_i \in \mathcal{D}} (Q_i) = Q$.

Step 3: Finally, we find the sets in $\mathcal{S}$ with the minimum number of subqueries and output one of them.

It is easy to see that the min-subquery algorithm returns a decomposition with minimum number of so-subqueries. Note that there might be multiple decompositions that minimize the number of so-subqueries.

Proposition 6.3.4. *Considering a query Q , the min-subquery algorithm results a decomposition $\mathcal{D}_Q$ of Q such that each query in $\mathcal{D}_Q$ is an so-query and $\mathcal{D}_Q$ has the minimum number of so-subqueries, among all the decompositions of Q including so-subqueries.*

The proof of the previous proposition follows by the Proposition 6.3.2. The last step of the algorithm also ensures that the output has the minimum number of so-subqueries.

Although the min-subquery algorithm returns a minimal decomposition, it applies an exhaustive search over the search space and the resulted decomposition has high redundancy (i.e., there are triples that are included in two subqueries). Especially, if the replicated edges include variables, as we saw in the previous Section, the amount of the intermediate results could affect the overall evaluation time. To overcome these issues, we focus on an efficient approach that constructs a decomposition based on the nodes' degree. In particular, we focus on selecting first the subqueries containing as many triples as possible. In addition, each query triple is included in a unique so-subquery (i.e. redundancy is not allowed in query decomposition). The decomposition algorithm, called *max-degree*, that follows this approach is given below.

```

max-degree( $Q$ )
//  $Q$  a query.
// max-degree function returns a decomposition  $R$  of  $Q$  consisting of so-subqueries
begin
   $R = \emptyset$ ;
   $N = \mathcal{N}(Q) - L$ ; // The non-literal nodes.
   $S_Q = \text{FindInitMaxSoQueries}(N)$ 
   $T_{\text{Covered}} = \emptyset$ ;
  while  $S_Q \neq \emptyset$  do
    begin
       $S_Q^{\text{old}}, R, T_{\text{Covered}} = \text{ReconstructS}_Q(S_Q, R, T_{\text{Covered}})$ ;
       $S_Q = \emptyset$ ;
      foreach  $(m, S) \in S_Q^{\text{old}}$  do
        begin
           $S' = S - T_{\text{Covered}}$ ;
          if there is a triple with subject  $m$  in  $S'$  then  $S_Q = S_Q \cup \{(m, S')\}$ ;
        end
      end
    end
  return  $R$ ;
end.

FindInitMaxSoQueries( $N$ )
//  $N$  is a set of the non-literal nodes of a query
begin
   $S_Q = \emptyset$ ;
  foreach  $n \in N$  do //  $S_Q$  contains all pairs  $(n, S(n))$  where  $n \in N$  and
    begin //  $S(n)$  is the maximal so-query with  $n$  as central node.
      if there is a triple  $(n, p, o) \in Q$  then
        begin
           $S(n) = \{t \mid t \in Q \text{ and } (t = (n, p, o) \text{ or } t = (s, p', n))\}$ ;
           $S_Q = S_Q \cup \{(n, S(n))\}$ ;
        end
      end
    end
  return  $S_Q$ ;
end.

ReconstructSQ( $S_Q, R, T_{\text{Covered}}$ )
// Find next so-subquery and update both the result  $R$  and the set  $T_{\text{Covered}}$  of covered edges.

```

```

begin
  //select a maximal so-query in  $S_Q$ 
  select a  $(n, Q') \in S_Q$  such that  $|Q'|$  is maximal among all elements in  $S_Q$ .
   $R = R \cup \{Q'\}$ ; // ... add  $Q'$  to the result and ...
   $T_{Covered} = T_{Covered} \cup Q'$ ; // ... add its triples to  $T_{Covered}$ .
   $S_Q^{old} = S_Q - \{(n, Q')\}$ ;
  return  $S_Q^{old}$ ,  $R$ ,  $T_{Covered}$ ;
end.

```

Intuitively, the max-degree algorithm performs similarly to the min-subquery algorithm. In particular, in each step, it finds an so-query with max degree and removes its edges from the remaining so-stars. The algorithm stops once all the query edges are covered. Note that the max-degree algorithm does not aim to minimize the number of subqueries in the decomposition. An example describing such a case is illustrated in Figure 6.4. Notice that D_1 and D_2 are two decompositions of Q , where D_1 is the result of the max-degree algorithm and D_2 is the result of min-subquery algorithm. On the other hand, there are queries where the results of both algorithms match. Such an example is illustrated in Figure 6.3.

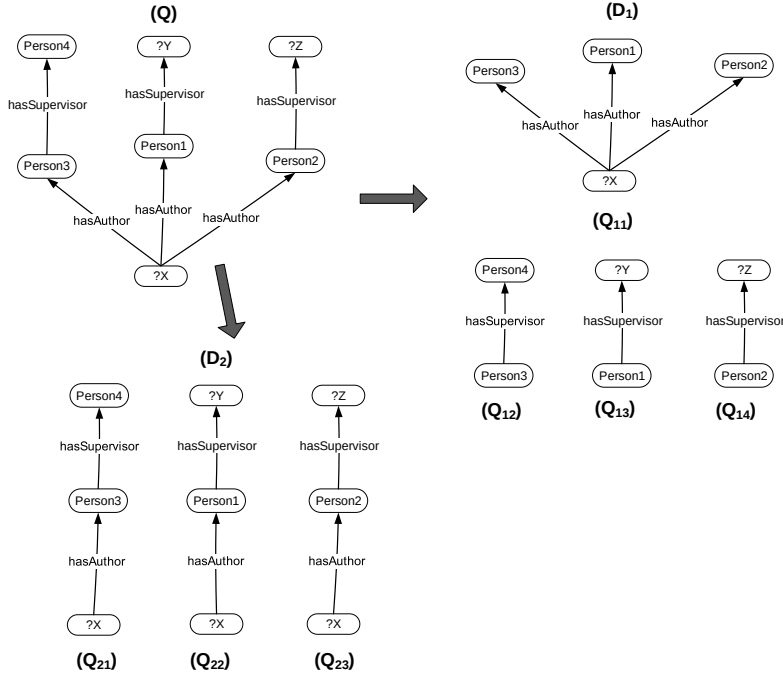


Figure 6.4: Min-subquery vs. Max-degree decomposition

As we mentioned above, the max-degree decomposition algorithm does not apply any edge replication (no redundant edges are allowed). Lack of replication might improve the performance of finding the embeddings of a subquery, since less edges are checked in order to find an embedding. However, as we saw in Proposition 6.3.1, replicating edges that add more constraints in the subquery might decrease the total number of embeddings of subqueries; hence, it may also decrease the communication cost in a distributed execution. Taking this into account, we present a modification of the max-degree algorithm, called *max-degree-with-redundancy*, which replicates edges with constants. The max-degree-with-redundancy algorithm is given by replacing the function *Reconstruct S_Q* in max-degree algorithm with the function *Reconstruct S_Q Redundancy*, which is defined as follows.

```

Reconstruct $S_Q$ Redundancy( $S_Q, R, T_{Covered}$ )
//Find next so-subquery and update both the result  $R$  and the set  $T_{Covered}$  of covered edges.
begin
  //select a maximal so-query in  $S_Q$ 
  select a  $(n, Q') \in S_Q$  such that  $|Q'|$  is maximal among all elements in  $S_Q$ .
  //add triples that have already covered and do not add any new variable to the subquery found
   $Q'' = Q' \cup \{t = (n_1, p, n_2) | t \in (Q - Q'), \text{ and either } n_1 = n \text{ and } n_2 \notin \mathcal{V}(Q) \text{ or } n_2 = n \text{ and } n_1 \notin \mathcal{V}(Q)\}$ ;
   $R = R \cup \{Q''\}$ ; // ... add  $Q''$  to the result and ...
   $T_{Covered} = T_{Covered} \cup Q'$ ; // ... add its triples to  $T_{Covered}$ .
   $S_Q^{old} = S_Q - \{(n, Q')\}$ ;
  return  $S_Q^{old}, R, T_{Covered}$ ;
end.
```

Comparing the max-degree and max-degree-with-redundancy algorithms, we can easily see that the function *Reconstruct S_Q Redundancy* used in the max-degree-with-redundancy algorithm to construct each subquery and add it into the resulting set R , constructs each subquery Q'' from Q' (which is similar to the query Q' constructed by *Reconstruct S_Q* in the max-degree algorithm) and all the query triples in Q that either start or end to the central node of Q' and do not include a variable in the other node; i.e., the triples having constants in the non-central node are replicated and reused. In the contrary, the max-degree algorithm (function *Reconstruct S_Q*) does not replicate any edge during construction of the result.

Proposition 6.3.5. *Considering a query Q , the results of both the max-degree and max-degree-with-redundancy algorithms are decompositions of Q that include only so-queries.*

Proof. By construction all subqueries produced in both algorithms are so-queries. Besides, as all query triples are used the algorithms produce decompositions of

Q .

□

As we mentioned above the main difference between max-degree and max-degree-with-redundancy algorithms is that in the latter, we replicate edges that have constants in the adjacent nodes of the central node. As Proposition 6.3.6 shows, the decomposition resulted by the max-degree-with-redundancy algorithm might reduce the number of embeddings exchanged between the last two steps of the evaluation algorithms eval-STARS and QE-with-Redundancy, comparing to the corresponding decomposition resulted by the max-degree algorithm.

Proposition 6.3.6. *Let Q be a query and $\mathcal{D}_M$ be a decomposition of Q resulted by the max-degree. Then, there is a decomposition $\mathcal{D}_R$ of Q resulted by the max-degree-with-redundancy algorithm such that the following hold:*

- *there is an one-to-one mapping μ from $\mathcal{D}_M$ to $\mathcal{D}_R$ such that $\mu(P_R) = P_M$, if $P_M \subseteq P_R$, where $P_R \in \mathcal{D}_R$ and $P_M \in \mathcal{D}_M$; and*
- *for each data graph G and every query P in $\mathcal{D}_R$, the number of embeddings of P over G is less than or equal to the number of embeddings of $\mu(P)$ over G .*

Proof. Let Q be a query and $\mathcal{D}_M$ be a decomposition of Q resulted by the max-degree. We now need to prove that the max-degree-with-redundancy algorithm can result a decomposition of Q which satisfies the aforementioned properties.

Each subquery Q' in $\mathcal{D}_M$ is constructed by the function $ReconstructS_Q$, and specifically, once it is constructed it is inserted into the resulting set R (which eventually equals $\mathcal{D}_M$). Let's consider that instead of returning the subquery Q' into the result R , we return the subquery $Q'' = Q' \cup \{t = (n_1, p, n_2) | t \in (Q - Q')\}$, and either $n_1 = n$ and $n_2 \notin \mathcal{V}(Q)$ or $n_2 = n$ and $n_1 \notin \mathcal{V}(Q)\}$, where n is the central node of Q' . Since R simply stores the resulting subqueries and is not used in any other step of the algorithm, such a modification does not affect the construction of the subqueries. It is easy to verify that the modified function is given by the function $ReconstructS_QRedundancy$, and the modified algorithm is the max-degree-with-redundancy. Let also $\mathcal{D}_R$ be the result of the modified algorithm (i.e., the final set R returned by the algorithm); hence, $\mathcal{D}_R$ is the result of the max-degree-with-redundancy.

According to the previous modification, for each query Q' in $\mathcal{D}_M$, there is a query Q'' in $\mathcal{D}_R$, such that $Q'' = Q' \cup \{t = (n_1, p, n_2) | t \in (Q - Q')\}$, and either $n_1 = n$ and $n_2 \notin \mathcal{V}(Q)$ or $n_2 = n$ and $n_1 \notin \mathcal{V}(Q)\}$, where n is the central node

of Q' . Hence, there is an one-to-one mapping μ from $\mathcal{D}_M$ to $\mathcal{D}_R$ such that $Q'' = \mu(Q')$ and $Q' \subseteq Q''$; which proves the first condition of the proposition.

Furthermore, Proposition 6.3.1 and the construction of Q'' from Q' imply that for each data graph G , the number of embeddings of Q'' over G is less than or equal to the number of embeddings of Q' over G ; which means that the second property is also satisfied. Hence, the decomposition $\mathcal{D}_R$ satisfies both properties of the Proposition 6.3.6. $\square$

As we saw previously, both max-degree and max-degree-with-redundancy algorithms iterate over the maximal so-subqueries found by the *FindInitMaxSoQueries* function, from the queries of max degree to the queries with min degree, and remove triples covered in the previous iterations. In each iteration, if the query resulted by removing the covered triples is not an so-query, then both the algorithms ignore this query and continue to the next iteration. Let Q_R be the set of the remaining triples, in such cases. Note that Q_R will be covered in the next iterations, but the number of iterations might increase due to the triples that do not form an so-query in some iterations. To reduce the number of iterations, we can construct so-queries from the triples in Q_R by adding to Q_R a triple that makes it so-query. Such a triple t is found in the set of covered triples. In addition, to avoid replicating triples that add variables to a query, we remove the triple t from the so-query that was constructed in the previous iterations. Such an approach might reshape the so-queries constructed in the previous iterations. A decomposition algorithm following this approach is given as follows, and called *max-degree-with-reshaping*.

```

max-degree-with-reshaping( $Q$ )
//  $Q$  a query.
// The function returns a decomposition  $R$  of  $Q$  consisting of so-subqueries of  $Q$ 
begin
   $R = \emptyset$ ;
   $N = \mathcal{N}(Q) - L$ ; // The non-literal nodes.
   $S_Q = \text{FindInitMaxSoQueries}(N)$ ;
   $T_{\text{Covered}} = \emptyset$ ;
  while  $S_Q \neq \emptyset$  do
    begin
      select a  $(n, Q') \in S_Q$  s.t.  $\forall (m, Q'') \in (S_Q - \{(n, Q')\})$  it holds  $NC(Q') \geq NC(Q'')$ ;
      //  $NC(Q)$  function returns the number of Not Covered triples in  $Q$ 
       $S' = Q' - \{t \mid t = (s, p, n) \in Q' \text{ and } t \in T_{\text{Covered}} \text{ and } s \in \mathcal{V}(Q')\}$ ;
      //i.e. remove covered triples whose object is  $n$  that add variable in  $Q'$ 
       $T = \{t \mid t = (n, p, o) \in Q' \text{ and } t \in T_{\text{Covered}} \text{ and } o \in \mathcal{V}(Q')\}$ ;
      If  $S' - T$  is so-query then  $S = S' - T$  else
        begin
           $S = S' - T \cup \{t'\}$  where  $t'$  is a triple in  $T$ ;
          replace  $F$  by  $F - \{t'\}$  in  $R$  where  $F$  is the query in  $R$  containing  $t'$ ;
        end
      end
    end

```

```

        // notice that  $F - \{t'\}$  is also an so-query
    end

     $R = R \cup \{S\}$ ; // ... add  $S$  to the set of subqueries of  $Q$  ...
     $T_{Covered} = T_{Covered} \cup S$ ; // ... add the triples of  $S$  to  $T_{Covered}$ .
     $S_Q^{old} = S_Q - \{(n, Q')\}$ ;
     $S_Q = \emptyset$ ;
    foreach  $(m, S) \in S_Q^{old}$  do //Reconstruct  $S_Q$  by removing the queries whose triples are ...
        begin //... completely covered by the so-queries already constructed ...
            if  $S - T_{Covered} \neq \emptyset$  then  $S_Q = S_Q \cup \{(m, S)\}$ ;
        end
    end
end
return  $R$ ;
end.

```

Proposition 6.3.7. *Considering a query Q , the results of max-degree-with-reshaping algorithm are decompositions of Q that include only so-queries.*

Proof. By construction all subqueries produced in the algorithm are so-queries. Besides, as all query triples are used the algorithm produces decompositions of Q . $\square$

6.4 Discussion

QE-with-Redundancy algorithm has several advantages compared with QEJPE-algorithm and eval-STARS algorithm. Notice that QE-with-Redundancy algorithm is implemented using one and a half Map-Reduce phases while QEJPE-algorithm and eval-STARS algorithm are implemented using two Map-Reduce phases. Another advantage of QE-with-Redundancy algorithm is that, due to the replication of the data triples in the decomposition of the data graph, and the special form of subqueries in which the user query Q is decomposed, namely subject-object star queries, all the answers to a subject-object star queries can be obtained from a single data segment.

On the other hand, due to the replication of the data triples in the decomposition of the data graph, multiple occurrences of the same embedding as well as multiple instances of member of MBL list may be produced and emitted in Phase 1 of the algorithm.

Chapter 7

Query Evaluation Over Linked Data Stored in a Document Database

In this Chapter, we propose a method for partitioning the data graph into graph segments so as every graph segment is stored as a different document in a *NoSQL document* database. Using the proposed partitioning method, the answers of every generalized star query can be computed in document level avoiding, thus, a sequence of computationally expensive joins over large data sets. The proposed method facilitates horizontal scaling in a balanced manner. Additionally we propose an algorithm, called Doc-based Algorithm, for query evaluation over LD stored in a document database. The Doc-based Algorithm is implemented in Apache Spark and the data is stored in MongoDB.

7.1 Preliminaries

In this Section we extend the definitions of Chapter 3 about the *query graph* and the *embeddings* to allow queries with variables in the place of predicates.

Definition 7.1.1. Let U_{so} and U_p be two disjoint infinite sets of URI references, let L be an infinite set of (plain) literals and let V and V_p be two disjoint infinite sets of variables. An element $(s, p, o) \in (U_{so} \cup V) \times (U_p \cup V_p) \times (U_{so} \cup L \cup V)$ is called a query triple. A query graph (or simply a query) is a labeled, directed

graph defined as a non-empty set of query triples. The output pattern $O(Q)$ of a query graph Q is the tuple $\langle (X_1, \dots, X_n), (P_1, \dots, P_m) \rangle$, where $n \geq 0$ and $m \geq 0$, and $X_1, \dots, X_n \in V$, and $P_1, \dots, P_m \in V_p$ and $X_1, \dots, X_n, P_1, \dots, P_m$ appear in Q . A query graph Q' is a subquery of a query graph Q if $Q' \subseteq Q$. A query consisting of a sequence of query triples of the form $(n_0, p_1, n_1), (n_1, p_2, n_2), \dots, (n_{k-1}, p_k, n_k)$, with $k \geq 1$, is said to be a chain query of length k .

Notice that there might be variables of Q that do not appear in its output pattern $O(Q)$. Concerning the literals, in this paper we do not consider typed literals. A query graph Q typically represents a BGP SPARQL query.

We also consider the following notation: The *nodes* $\mathcal{N}(G)$ (resp. $\mathcal{N}(Q)$) of a data graph G (resp. a query Q) is the set of elements of $U_{so} \cup L$ (resp. $U_{so} \cup L \cup V$) that occur in the triples of G (resp. Q). The set of literals of G (resp. Q) is denoted as $\mathcal{L}(G)$ (resp. $\mathcal{L}(Q)$). The set of edges of G (resp. Q) is denoted as $\mathcal{E}(G)$ (resp. $\mathcal{E}(Q)$). The *edge labels* $\mathcal{L}_{\mathcal{E}}(G)$ (resp. $\mathcal{L}_{\mathcal{E}}(Q)$) of a data graph G (resp. a query Q) is the set of elements of U_p (resp. $U_p \cup V_p$) that occur in the triples of G (resp. Q).

To compute the answers of a query Q for a given data graph G , we consider finding an appropriate mapping from the nodes and edges of Q to the nodes and edges of G respectively. This is formalized by the concept of *embedding* defined as follows.

Definition 7.1.2. An embedding of a query graph Q to a data graph G is a total mapping $e : \mathcal{N}(Q) \cup \mathcal{L}_{\mathcal{E}}(Q) \rightarrow \mathcal{N}(G) \cup \mathcal{L}_{\mathcal{E}}(G)$ satisfying the following conditions:

1. For each node $v \in \mathcal{N}(Q)$, if $v \in V$ then $e(v) \in \mathcal{N}(G)$ otherwise $e(v) = v$.
2. For each edge label $p \in \mathcal{L}_{\mathcal{E}}(Q)$, if $p \in V_p$ then $e(p) \in \mathcal{L}_{\mathcal{E}}(G)$ otherwise $e(p) = p$.
3. For each triple $(s, p, o) \in Q$, the triple $(e(s), e(p), e(o))$ is in G .

We denote as $e(Q)$ the set of triples: $\{t | t = (e(s), e(p), e(o)) \text{ and } (s, p, o) \in Q\} \subseteq G$. The result of Q on G , denoted as $Q(G)$, is the set $\{\langle (e(X_1), \dots, e(X_n)), (e(P_1), \dots, e(P_m)) \rangle | \langle (X_1, \dots, X_n), (P_1, \dots, P_m) \rangle \text{ is the output pattern of } Q, \text{ and } e \text{ an embedding from } Q \text{ to } G\}$. Each tuple in $Q(G)$ is an answer of Q .

Additionally, in this Chapter we only use generalized star queries that the central node c is not a literal.

Definition 7.1.3. A query Q is called a generalized star query if there exists a node $c \in \mathcal{N}(Q)$ such that for every triple $t \in Q$, c is either the subject or the object of t and c is not a literal. In all cases, c is called the central node of Q and is denoted as $C(Q)$.

Example 7.1.1. Five generalized star queries are depicted in Fig. 7.1. It is easy

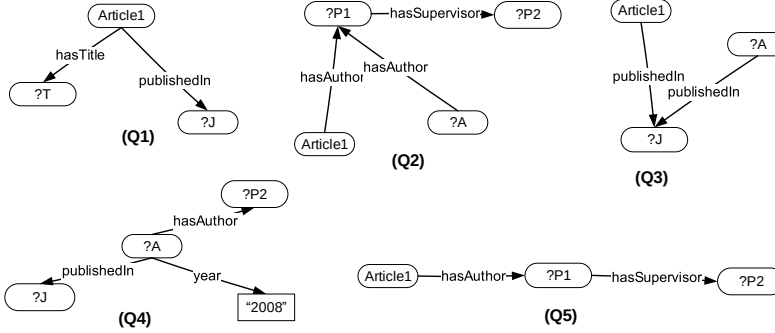


Figure 7.1: Examples of generalized star queries.

to see that all these queries are subqueries of the query Q appearing in Fig. 3.1. Only the queries $Q1$ and $Q4$ are star queries as defined in [33, 125] while the query $Q5$ is also a chain query [33, 125] of length 2.

7.2 The Node-oriented Partitioning of a Data Graph

A data graph may consist of a large amount of data triples; hence, storing the data into a distributed system might be required for effective and efficient data management. In this Section, we propose a method for partitioning the data graph into graph segments in a way that every graph segment is stored as a different document in a NoSQL/document database. Using the proposed partitioning method, the answers of every generalised star query can be computed in document level; avoiding, in such a way, a sequence of computationally expensive joins over large data sets. The proposed method facilitates horizontal scaling in a balanced manner.

The definition of the partition scheme, called *node-oriented partition* (or simply *node-partition*), is given as follows.

Definition 7.2.1. Let G be a data graph. A node-oriented partition (or node-partition for short) of G is a set of subgraphs $\mathcal{P}(G) = \{G_{n_1}, \dots, G_{n_m}\}$ of G , with $n_i \in (\mathcal{N}(G) - \mathcal{L}(G))$ and $1 \leq i \leq |\mathcal{N}(G) - \mathcal{L}(G)| = m$, such that for each $n_i \in (\mathcal{N}(G) - \mathcal{L}(G))$, with $1 \leq i \leq m$, $G_{n_i} = \{t \mid t = (s, p, o) \in G \text{ and either } s = n_i \text{ or } o = n_i\}$. Subgraphs $G_{n_1}, \dots, G_{n_m}$ are called (node-)graph segments. The node n_i node is called central node of G_{n_i} .

The above definition implies that $\bigcup_{i=1}^m G_{n_i} = G$, where the set-union operator is considered (i.e., duplicate triples are eliminated). Besides, we can easily see that each triple whose object is URI appears in two node-graph segments, while each triple whose object is literal appears in a single node-graph segment. In particular, consider the triple $t = (s, p, o)$, and assume that o is a URI (notice that s is always a URI). Then t belongs to two graph segments; i.e., the graph segments G_s and G_o . Then, the occurrence of t in G_o is said to be a *replicated triple* of G_o .

The following proposition gives an upper bound of the size of data in a node-oriented partition.

Proposition 7.2.1. Let G be a data graph, and $\mathcal{P}(G) = \{G_{n_1}, \dots, G_{n_m}\}$, with $m \geq 1$, be a node-partition of G . Considering $S = \sum_{i=1}^m |G_{n_i}|$, we have that $S \leq 2 * |G|$.

Example 7.2.1. Nine different node-graph segments, one for each different URI node of the data graph, are obtained by applying the node-partition method on the data graph of Fig. 3.1(a). Two of these node-graph segments are depicted in Fig. 7.2 and are given below. Notice that, the triple (Article1,hasAuthor,Person2) appears in both of them:

Segment G1 (with central node Article1):

$\{(Article1, hasAuthor, Person1), (Article1, hasAuthor, Person2), (Article1, hasAuthor, Person4), (Article1, hasTitle, "Title1"), (Article1, year, "2005"), (Article1, publishedIn, Journal1)\}$.

Segment G2 (with central node Person2):

$\{(Article1, hasAuthor, Person2), (Article2, hasAuthor, Person2), (Person2, hasSupervisor, Person3)\}$.

Theorem 7.2.1. Let $\mathcal{P}(G) = \{G_1, \dots, G_m\}$, with $m \geq 1$, be a node-partition of a data graph G , and Q be a generalized star query. Then, a mapping e is an embedding from Q to G if and only if there is a node graph segment G_e in $\mathcal{P}(G)$ such that $e(Q) \subseteq G_e$. Consequently, $Q(G) = \bigcup_{G_i \in \mathcal{P}(G)} Q(G_i)$.

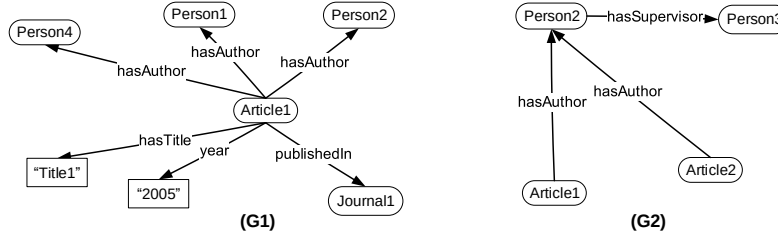


Figure 7.2: Two node-partition segments G1, G2 obtained from the graph G of Fig. 3.2.

Proof. Consider a mapping e from Q to G . The image $e(C(Q))$ of the central node $C(Q)$ of Q in G is a node n_c in G such that n_c is a URI. Then n_c is the central node of node-graph segment G_{n_c} in $\mathcal{P}(G)$. By the construction of the graph node segments and the assumption that Q is a generalized star query, we see that all the triples that are images of the triples of Q through e , are triples in G_{n_c} . The opposite direction is straightforwardly implied by the definitions of node-partition and generalized star queries. $\square$

Theorem 7.2.1 shows that the evaluation of a (generalized) star query Q over a segment G_i of a node-partition $\mathcal{P}(G)$ of a data graph G is independent of the evaluation of Q over the other segments in $\mathcal{P}(G)$. Based on this property, we present, in Section 7.5, an efficient approach for evaluating such queries over large data graphs, using a distributed document database (MongoDB).

Example 7.2.2. (Continued from Example 7.2.1) Applying the star query Q_5 depicted in Fig. 7.1 to the graph segment G_2 we get the answer: $\langle (?P1, ?P2), () \rangle = \langle (Person2, Person3), () \rangle$.

7.3 Decomposing Query Graphs into Generalized Star Queries

In this Section, we propose a methodology for decomposing an arbitrary graph query into generalized stars. In addition, our aim is to construct the embeddings of a query Q in G , by properly combining embeddings of its (generalized star) subqueries in G . Two embeddings can be combined only in the case that they

match in the values of the nodes and edges that are common in the corresponding subqueries. The above requirement is formalized by the following definition:

Definition 7.3.1. Let Q_1, Q_2 be two query graphs and let e_1, e_2 be embeddings of Q_1, Q_2 respectively in a data graph G . We say that e_1 and e_2 are compatible if for every (node) variable $v \in \mathcal{N}(Q_1) \cap \mathcal{N}(Q_2) \cap V$ it is $e_1(v) = e_2(v)$ and for every (edge label) $l \in \mathcal{L}_E(Q_1) \cap \mathcal{L}_E(Q_2) \cap V_p$ it is $e_1(l) = e_2(l)$. The join of e_1 and e_2 is the embedding e of $Q_1 \cup Q_2$ in G defined as follows:

$$e(v) = \begin{cases} e_1(v) & \text{if } v \in \mathcal{N}(Q_1) \\ e_1(v) & \text{if } v \in \mathcal{L}_E(Q_1) \\ e_2(v) & \text{otherwise} \end{cases}$$

It is easy to see that the join operation is commutative and associative. Therefore, we can refer to the embedding resulted by the join of n mutually compatible embeddings without ambiguity.

The notions of *query decomposition* and *border nodes* are defined in the same way as in Section 3.3

The following theorem relates the embeddings of a query Q to the embeddings of its subqueries in a decomposition of Q .

Theorem 7.3.1. Let $\mathcal{D}_Q = (Q_1, \dots, Q_n)$ be a query decomposition of a query graph Q and let G be a data graph. Then e is an embedding of Q in G if and only if there exist mutually compatible embeddings $e_1, \dots, e_n$ of $Q_1, \dots, Q_n$ in G such that the join of $e_1, \dots, e_n$ is e .

Proof. If e is an embedding of Q in G then, by the definition of query decomposition, for each query Q_i we get an embedding e_i of Q_i in G by restricting e to the mapping of nodes and edges of Q_i ; i.e., for each node n of Q_i we have $e_i(n) = e(n)$. For the other direction, it is easy to see (based on Definition 7.3.1) that if $e_1, \dots, e_n$ are compatible embeddings of $Q_1, \dots, Q_n$ in G , respectively, then the join e of $e_1, \dots, e_n$ is an embedding of Q in G . $\square$

The above theorem implies that in order to compute the answers of a given query over a data graph G , we can decompose the query into subqueries that belong to a certain class C , compute the embeddings of the subqueries in G (which may be an easier task due to the special form of the subqueries) and then join these embeddings to obtain the desired result. However, given a target class of queries C , it may not be possible to decompose an arbitrary query Q into

subqueries that belong to C (for example this is the case if C is the class of path queries of length 3, which are of the form $\{(u, p, v), (v, p', w), (w, p'', z)\}$).

Nevertheless, if C is the class of generalized star queries the following proposition holds:

Proposition 7.3.1. *Every query Q can be decomposed into a set of generalized star subqueries.*

This proposition is trivially implied by the fact that every query that consists of a single triple is a generalized star query (with either the subject or the object being the central node).

The following theorem relates the embeddings of a query Q on a graph G to the embeddings of the generalized star subqueries of a decomposition of Q over the node-graph segments obtained by a node-oriented partition of G .

Theorem 7.3.2. *Let $\mathcal{D}_G = \{G_1, \dots, G_m\}$, with $m \geq 1$, be a node-partition of a data graph G , and $\mathcal{D}_Q = (Q_1, \dots, Q_n)$ be a query decomposition of a query graph Q such that $Q_1, \dots, Q_n$ are generalized star queries. Then e is an embedding of Q in G if and only if there exist mutually compatible embeddings $e_1, \dots, e_n$ of $Q_1, \dots, Q_n$ in $G'_1, \dots, G'_n$ respectively, where $G'_i \in \mathcal{D}_G$, with $1 \leq i \leq n$, such that the join of $e_1, \dots, e_n$ is e .*

Proof. If e is an embedding of Q in G then for each query Q_i we get an embedding e_i of Q_i in G by restricting e to the mapping of nodes and edges of Q_i in G . It is easy to see by construction that $e_1, \dots, e_n$ are mutually compatible. Moreover, from Theorem 7.2.1 we conclude that for each Q_i there is a node graph segment G'_i in $\mathcal{D}_G$ such that e_i is an embedding of Q_i in G' . For the other direction, assume that there are mutually compatible embeddings $e_1, \dots, e_n$ of $Q_1, \dots, Q_n$ respective in $G'_1, \dots, G'_n$, where $G'_i \in \mathcal{D}_G$. It is easy to see that the join e of $e_1, \dots, e_n$ is an embedding of Q in G . $\square$

Note here that there are multiple approaches to decompose an arbitrary query Q into generalized star queries; e.g., maximize triples in subqueries, or maximize the number of constants per subquery, or increase the triple replication across queries. In this Section, we propose a decomposition algorithm, appearing in Fig. 7.3, focusing on maximizing the number of triples in each (generalized star) subquery, minimizing the replication of variables into multiple subqueries and taking advantage of literals for filtering the intermediate results. Redundancy is allowed between the subqueries obtained by applying the algorithm.

More specifically, the algorithm is based on the following approach: For each non-literal node n in Q we construct a generalized star query containing all incoming and outgoing edges of n . We then prune systematically the set of these generalized star queries to construct the final set of generalized star queries.

In particular:

1. Subqueries containing as many as possible uncovered (i.e. non replicated) triples of Q are selected first. In this way, the number of subqueries produced by decomposing the query Q is kept as small as possible. Such selection is based on the observation that as the number of the query triples is increasing the number of results (number of documents matching the subquery) is decreasing.
2. Among the subqueries with equal number of uncovered triples the subqueries whose central nodes are URIs precede to our selection; comparing with the subqueries whose central nodes are variables.
3. In case that the above criteria are satisfied by more than one generalized star queries we select the query with the maximum number of literals and URI nodes.

The above criteria are applied iteratively until all triples of the query Q are covered by (are contained in) at least one generalized star subquery. It is easy to see that applying the above procedure, some of the triples of the initial query Q may be contained in more than one subqueries.

7.4 The Doc-based Algorithm

The node-oriented partition of a data graph G presented in Section 7.2 and the query decomposition of a query Q into generalized star queries presented in Section 7.3 are taken into consideration in the following query evaluation strategy; which can be utilized for evaluating graph queries over distributed data graphs, in parallel.

1. The data graph G is partitioned using the node-oriented partitioning approach proposed in Section 7.2.
2. The query graph Q is decomposed into a set $\mathcal{D}_Q$ of generalized star subqueries by applying the algorithm presented in Fig. 7.3 and analyzed in Section 7.3.

3. We, then, find the embeddings of each subquery in $\mathcal{D}_Q$ on each node-graph segment.
4. Finally, we join the compatible embeddings, one for each subquery, in order to construct the embeddings of the query Q .

7.5 Doc-based Implementation using MongoDB and Apache Spark

In this Section we propose a data model for storing node-oriented partitioned data into a document database, and specifically, into MongoDB. Although the proposed data model could be applied in most of the NoSQL databases that support document model for structuring the data, the approach is focused on MongoDB, due to proven high-performance management of large amount of hierarchical data and support of advanced indexing strategies (e.g., multikey indexes).

We also present a simple distributed algorithm for constructing such a partition over the data graph stored into a distributed file system. The purpose of this algorithm is to efficiently partition a large collection of RDF triples and load the partitioned data into a MongoDB, properly (complying with the proposed data model).

Considering that the data graph is properly partitioned (i.e., node-oriented partitioned) and stored into a MongoDB collection, the evaluation of an arbitrary query Q over the partitioned data is performed in two phases. The first phase is to properly query the data stored in MongoDB (in order to extract the data required to answer the query Q) and the second phase focuses on efficiently combining the intermediate results extracted in the first phase in order to provide the answer of Q . In order to identify and extract, in phase 1, only the data required for answering query Q , we firstly apply the query decomposition approach presented in Section 7.3. Then, we use the subqueries in the decomposition of Q in order to efficiently extract, from the MongoDB, the required data used to construct the answer of Q . Based on the proposed data model, we also provide a simple algorithm which translates the graph subqueries into the corresponding MongoDB queries. The results of the subqueries are loaded into Spark in order to be combined and result the answer of Q . Subsection 7.5.3 describes, in more detail, the overall evaluation approach.

7.5.1 Document Data Model for Storing Node-Oriented Data Graph Decomposition Segments

In this Section, we present an approach of utilizing the node-partition technique defined in Section 7.2, for efficient evaluation of generalized star queries over large data graphs. The proposed model for storing a node-partitioned data graph preserves the directed graph structure of the data into each node-graph segment. In particular, each node-graph segment is stored in hierarchical/document structure (each segment is translated to a single document).

The data model used to store each node-graph segment in a different document is defined in Fig. 7.4(a). In this model, each segment document/object consists of the following attributes:

1. the URI of the central node
2. an array of subjects including all the outgoing edges
3. an array of objects including all the incoming edges

Note that we consider grouping of all the incoming and outgoing edges according to their predicate in order to reduce the replication of the predicate values in the edges.

As an example, in Fig. 7.4(b), we can see the JSON document file that represents the node-graph segment G_2 depicted in Fig. 7.2.

7.5.2 Efficient Parallel Partitioning Algorithm

To ensure efficient node-partitioning of data graphs and loading the resulted segments into MongoDB, we propose the distributed, MapReduce algorithm illustrated in Fig. 7.5. The algorithm works as follows: The *map function* gets an input file containing data triples and for each triple outputs: (a) a pair including the subject as key and a tuple including the predicate and object as value (along with a flag specifying that the value corresponds to outgoing edge), and, (b) if the object is not literal, a pair including the predicate and subject as value (along with a flag specifying that the value corresponds to incoming edge). *Reduce function* gets all triples of a certain node-segment and distinguish them into a group of outgoing and a group of incoming edges. Within each group, it applies a secondary grouping based on the predicate values. From these groups the reduce task generates a JSON document which represents a node-segment of the

data graph according to the data model presented in Fig. 7.4(a). Finally, the reducers upload/store the constructed documents into MongoDB.

The aforementioned algorithm could be straightforwardly translated into Spark as follows:

```
starRDD = triplesRDD
          .flatMap(triple: createStarCenters(triple))
          .groupByKey()
          .map(transform2MongoDoc)
```

where *triplesRDD* represents the triples RDD initialized by loading the triples into Spark and *starRDD* represents the resulted star-partitioned data. The function *createStarCenters* generates, for each triple, two key-value pairs where the triple's subject and object are the keys, respectively, and the value is given by the triple itself. *transform2MongoDoc* is a function that transforms, for each central node, the star data subgraph (built by the grouping) into a document-like structure which is acceptable by the Spark-MongoDB Connector.

7.5.3 Evaluating Graph Pattern Queries using MongoDB and Apache Spark

Considering a node-partitioned data graph stored in a MongoDB collection, as analyzed in previous Sections, it is easy to verify that such a partitioning approach ensures low-latency query evaluation of generalized star queries. The main reason for this statement is that each query is evaluated over each document, without requiring any time-consuming join operation. In addition, due to horizontal-scaling feature of MongoDB, we could achieve high-performance query evaluation over huge amount of data distributed in a Mongo cluster. The challenge arises when the given query expression is not limited to a generalized star query.

In this Section, we present an approach for efficiently evaluate arbitrary queries over a node-partitioned data graph. In particular, the approach is based on decomposing the given query graph into generalized star queries. These queries are evaluated over the node-partitioned data, stored in a distributed collection. The results of those queries are combined through a Spark DAG process, which ensures computation of the answer of the given query. The overall process is managed by a driver application which ensures communication between MongoDB and Spark, as well as management of the overall process.

The overall evaluation algorithm is described by the following steps. Consider a graph query Q , given as SPARQL expression, over a data graph G . We also

assume that the graph G is partitioned, according to node-partitioning approach presented in Section 7.1, and stored in a MongoDB collection C_G using the algorithm presented in Subsection 7.5.2. The data stored in MongoDB conforms to the data model presented in Subsection 7.5.1.

Step 1: Initially, the query graph Q is decomposed into a set $\mathcal{D}_Q$ of generalized star subqueries by applying the algorithm presented in Fig. 7.3.

Step 2: Each subquery in $\mathcal{D}_Q$ is transformed into the corresponding MongoDB query over C_G , using the algorithm presented in Fig. 7.6. The resulting set of MongoDB queries is denoted $\mathcal{DM}_Q$.

Example 7.5.1. *The MongoDB query constructed by applying the aforementioned transformation algorithm to the generalized star query Q_2 depicted in Fig. 7.1 is:*

```
{ "$and" : [
  { "object.predicate" :
    { "$elemMatch": { "value" : "hasAuthor" ,
      "subject" : { "$elemMatch" : {
        "value": "Article1"
      } } } } },
  { "object.predicate" :
    { "$elemMatch" : { "value": "hasAuthor" } } },
  { "subject.predicate" :
    { "$elemMatch" : { "value": "hasSupervisor" } } }
]
```

Step 3: All the queries in $\mathcal{DM}_Q$, are posed into C_G , in parallel, and return the documents (i.e., the star segments of G) s.t., there is at least one embedding from a query in $\mathcal{D}_Q$. Note that the result of each subquery defines a certain RDD into Spark (one for each query in $\mathcal{D}_Q$).

Step 4: Then, the RDDs created in the previous step are flattened, properly, and translated into relational-like structures; since the RDDs created in the previous step are given in nested form (implied by the nested document structure). In particular, for each query S in $\mathcal{D}_Q$, we create a relational structure having one field for each node of S . Each record of this structure corresponds to an embedding of S over the corresponding graph segment; e.g., if e is an embedding from S to the segment G_i , for each node n in $\mathcal{N}(S)$, there is a field in the S 's relation with name n and value $e(n)$. Note that this step is implemented in Spark and no data shuffling between cluster nodes is required (i.e., such a transformation is performed in parallel over each element of the RDD).

Step 5: The relational-like RDDs created in the previous step are then translated into Spark DataFrames, in order to utilize the optimized join operations provided by Spark for managing DataFrames. Note that for each subquery corresponds a single DataFrame.

Step 6: Finally, the DataFrames built in the previous step (one for each subquery) are joined over the common queries nodes/fields in order to compute the answer of the initial query posed into the data graph.

Note that the Steps 1 - 4 are performed in parallel without any communication cost required between cluster nodes. Only the Step 5 requires data shuffling, in order to compute the join operations between the DataFrames computed in Step 4. If the initial query is a generalized star query (i.e., no query decomposition is required), the overall query evaluation is completely performed in parallel, without any communication between the cluster nodes. If, however, the given query belongs to a super-class of generalized star queries, a sequence of join operations is required. Although the order the join operations are applied might provide additional improvement of the performance of the overall query evaluation, in this work, we rely on the Spark query planner, in order to construct an efficient query plan, and consider finding an optimal query plan as a topic for future investigation.

For example, Fig. 7.7 illustrates a diagram showing the overall evaluation process of a query Q which is decomposed into 3 subqueries (subQuery1, subQuery2 and subQuery3). The resulting RDDs constructed in Step 3 and resulted by querying MongoDB collection are named RDDJ_1, RDDJ_2 and RDDJ_3, respectively (one for each subquery), while the flattened RDDs constructed in Step 4 are named RDDM_1, RDDM_2 and RDDM_3, respectively. The flattened RDDs are respectively translated into the DataFrames DF_1, DF_2 and DF_3, as described in the Step 5. The last steps of the diagram show a potential join plan produced by Spark, in order to compute Q 's embeddings, and consequently return Q 's answer.

7.6 Discussion

RDF data can be efficiently stored on a document database through partitioning the overall data-graph into star segments. The proposed data partitioning approach ensures high-performance query answering for a wide sub-class of queries, called generalized star queries. The efficiency of the approach is based

on the feature that no joining operations are required between different data collections in order to completely evaluate a generalized star query over each data record/segment, and the segments could be evenly distributed in multiple cluster nodes.

Our approach is then extended to support the evaluation of general BGP SPARQL queries. The evaluation of these queries is based on the fact that every BGP query can be decomposed into a set of generalized star queries which can be evaluated independently of each other (and in parallel), and then the results of this evaluation can be appropriately combined to get the answers of the initial BGP query. Experimental evaluation shows that our approach is scalable in terms of the size of the data graph and efficient in the query evaluation.

As future work, we aim to investigate optimization of the query plans used to combine the results of the generalized star sub-queries. Furthermore, we aim to analyze additional query decomposition approaches, focusing on finding an optimal query decomposition for our setting. An additional topic for further investigation is how our approach could be extended to support query evaluation over dynamic RDF data.

```

max-subquery-with-redundancy-algorithm( $Q, R$ )
//  $Q$  a query.
//  $R$  a decomposition of  $Q$  consisting of generalized star subqueries of  $Q$ 
begin
   $R = \emptyset$ ;
   $N = \mathcal{N}(Q) - L$ ; // The non-literal nodes.
   $S_Q = \emptyset$ ;
  //  $S_Q$  aims to contain all the pairs  $(n, S(n))$  where  $n \in N$  and
  foreach  $n \in N$  do
    //  $S(n)$  is the maximal generalized star query with  $n$  as central node.
    begin
       $S(n) = \{t \mid t \in Q \text{ and } (t = (n, p, o) \text{ or } t = (s, p', n))\}$ ;
       $S_Q = S_Q \cup \{(n, S(n))\}$ ;
    end
   $T_{Covered} = \emptyset$ ;
  while  $S_Q \neq \emptyset$  do
    begin
      Let  $S_Q^{max} = \{(n, Q') \mid (n, Q') \in S_Q \text{ and } \forall (m, Q'') \in (S_Q - \{(n, Q')\}) NC(Q') \geq NC(Q'')\}$ 
      //  $NC(Q)$  function returns the number of Not Covered triples in  $Q$ 
      if  $S_Q^{max}$  is a singleton (i.e.  $S_Q^{max} = \{(n, Q')\}$ ) then  $S = (n, Q')$ 
      else
        begin
          Let  $S_Q^{URI} = \{(n, Q') \mid (n, Q') \in S_Q^{max} \text{ and } n \text{ is URI}\}$ 
          if  $S_Q^{URI}$  is a singleton (i.e.  $S_Q^{URI} = \{(n, Q')\}$ ) then  $S = (n, Q')$ 
          else
            begin
              if  $S_Q^{URI} = \emptyset$  then  $Temp = S_Q^{max}$ 
              else  $Temp = S_Q^{URI}$ 
              Let  $S_Q^{minvars} = \{(n, Q') \mid (n, Q') \in Temp \text{ and } \forall (m, Q'') \in (Temp - \{(n, Q')\}) |vars(Q')| \leq |vars(Q'')|\}$ 
              Select an element  $(n, Q')$  from  $S_Q^{minvars}$  and assign it to  $S$ 
            end
          end
        end
       $R = R \cup \{Q'\}$ ; // ... add  $Q'$ , where  $S = (n, Q')$  to the result and ...
       $T_{Covered} = T_{Covered} \cup Q'$ ; // ... add its triples to  $T_{Covered}$ .
       $S_Q^{old} = S_Q - \{(n, Q')\}$ ;
       $S_Q = \emptyset$ ;
    //Reconstruct  $S_Q$  by removing the queries whose triples are ...
    foreach  $(m, S) \in S_Q^{old}$  do
      //... completely covered by the generalized star queries already constructed ...
      begin
        if  $S - T_{Covered} \neq \emptyset$  then  $S_Q = S_Q \cup \{(m, S)\}$ ;
      end
    end
  end.

```

Figure 7.3: An algorithm for decomposing a query graph into generalized star queries.

<pre> object node-segment { string "centralNode"; object "subject" { string "value"; array of objects "predicate" { string "value"; array of objects "object" { string "value"; } } } } object "object" { string "value"; array of objects "predicate" { string "value"; array of objects "subject" { string "value"; } } } </pre>	<pre> { "centralNode" : "Person2", "subject" : { "predicate" : [{ "value" : "hasSupervisor", "object" : [{ "value" : "Person3" }] }] }, "object" : { "predicate" : [{ "value" : "hasAuthor", "subject" : [{ "value" : "Article1" }, { "value" : "Article2" }] }] } } </pre>
(a)	(b)

Figure 7.4: (a) Document data model, (b) Segment represented by a JSON document.

```

mapper(key, values)
// key: data file
// values: data triples  $t=(s, p, o)$ 
begin
  foreach data triple  $t$  in data file do
    Emit  $\langle s, ("subject", p, o) \rangle$ ;
    if ( $o \notin L$ ) then
      Emit  $\langle o, ("object", p, s) \rangle$ ;
    endif
end.

reducer(key, values)
// key:  $s$  and  $o \notin L$  of all data triples
// values: data triples  $t=(s,p,o)$ 
begin
  - Group the values per "object" and "subject";
  - For each "subject" group, group the values per predicate ( $p$ ) value;
  - For each "object" group, group the values per predicate ( $p$ ) value;
  - Construct the JSON document  $\mathbf{d}$  based on data model shown in Fig. 7.4(a);
  - Load  $\mathbf{d}$  using the corresponding MongoDB connector;
end.

```

Figure 7.5: A node-partition algorithm.

```

generalized star to MongoDB query transformation
// input: generalized star query  $Q$ 
// output: MongoDB query
begin
  Let  $V$  be the set of variables in  $Q$  and  $c=C(Q)$ ;
  if  $c \notin V$  then
    query += { centralNode :  $e(c)$  };
  foreach query triple  $t_i = (s_i, p_i, o_i)$  in  $Q$  do
    if  $s_i = c$  then
      if  $p_i \notin V$  and  $o_i \notin V$  then
        query += { subject.predicate : { $elemMatch :
          { value:  $e(p_i)$ , object : { $elemMatch : { value :  $e(o_i)$  } } } } };
      else if  $p_i \notin V$  and  $o_i \in V$  then
        query += { subject.predicate : { $elemMatch : { value :  $e(p_i)$  } } };
      else if  $p_i \in V$  and  $o_i \notin V$  then
        query += { subject.predicate.object : { $elemMatch : { value :  $e(o_i)$  } } };
    else
      if  $p_i \notin V$  and  $s_i \notin V$  then
        query += { object.predicate : { $elemMatch : { value :  $e(p_i)$ , subject :
          { $elemMatch : { value :  $e(s_i)$  } } } } };
      else if  $p_i \notin V$  and  $s_i \in V$  then
        query += { object.predicate : { $elemMatch : { value :  $e(p_i)$  } } };
      else if  $p_i \in V$  and  $o_i \notin V$  then
        query += { object.predicate.subject : { $elemMatch : { value :  $e(s_i)$  } } };
    MongoDBQuery = { $ and : [ + query + ] };
end.

```

Figure 7.6: An algorithm for transforming generalized star query to MongoDB query.

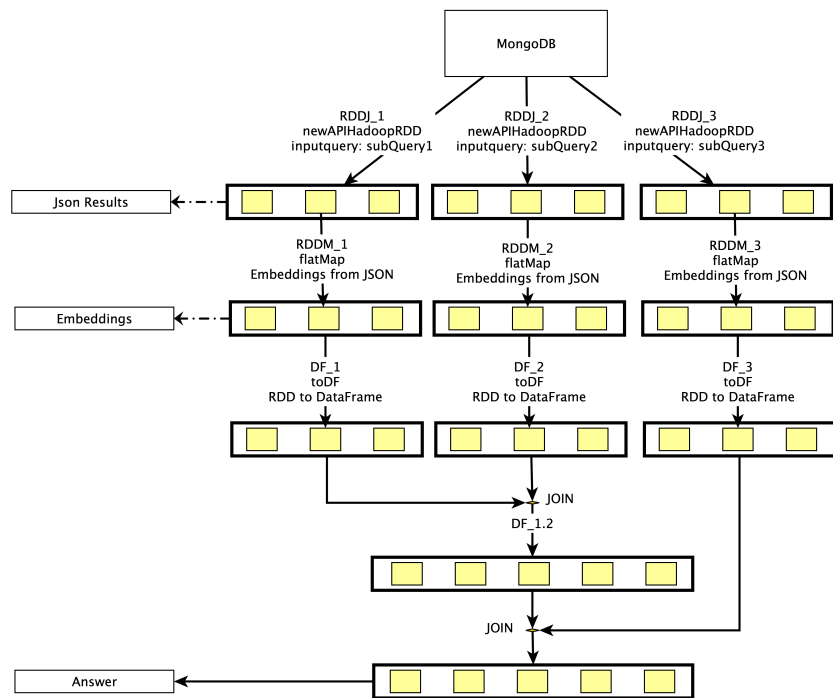


Figure 7.7: The DAG operations for the full query processing. An example with a query decomposed into three generalized star subqueries.

Chapter 8

Experimental Results

In this Chapter, we present a set of experiments that evaluate and compare the three MapReduce algorithms QEJPE, eval-STARS and QE-with-Redundancy described in Chapter 4, Chapter 5 and Chapter 6 respectively and the Doc-based algorithm presented in Chapter 7. Experiments results for the query decomposition algorithms, described in Section 6.3, are also presented.

8.1 Evaluation of the QEJPE, eval-STARS and QE-with-Redundancy Algorithms

To evaluate the QEJPE Algorithm, eval-STARS Algorithm and QE-with-Redundancy Algorithm experimentally and to compare the algorithms we present a set of experiments performed over a cluster of 10 virtual machines, and analyze the outcomes. Each cluster node has the following characteristics: Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz (8 Cores) with 16GB RAM, 60GB HD, Ubuntu 16.04 LTS, 64-bit Operating System. We used Apache Hadoop v3.1 with HDFS (1 NameNode, 1 Secondary NameNode, 10 DataNodes each one 30GB) and YARN (1 ResourceManager, 10 NodeManagers). The 10 virtual machines were connected through external IP addresses.

To perform the experiments we used four different datasets (D1, D2, D3, D4) in N-Triples format from the Waterloo SPARQL Diversity Test Suite (WatDiv) [33, 24]. The number of triples of each dataset, as well as the scale factors used to generate the datasets, are illustrated in Table 8.1.

The data graph was partitioned using three approaches, random edge par-

Dataset/Query	Scale Factor	Number of triples	Number of files
D1	25	2,731,510	7
D2	50	5,486,199	13
D3	100	10,979,566	25
D4	200	21,961,070	49

Table 8.1: Description of the Datasets

tition, vertex partition, and METIS. In particular, the random edge partition was implemented by randomly adding each edge into a file such that each file had approximately 450,000 triples. We also stored information about the border nodes in each file. The vertex partition essentially describes the implementation of s-decomposition approach defined in Chapter 6. The last partitioning approach used is METIS [75], in order to minimize the number of border nodes in each file. Note that the random edge and METIS partitioning approaches were used to evaluate queries using the QEJPE-algorithm and eval-STARS algorithms, while the vertex partition was used to evaluate the QE-with-Redundancy algorithm.

In the implementation of each algorithm, we used the library RDFLib¹ to pose the subqueries over data segments in each MapReduce task. In particular, in order to find the partial embeddings in each MapReduce task, we parse the data segment and load it into certain structure using RDFLib. Then, we use the query evaluation mechanism of the library to query the loaded segment and find the corresponding partial embeddings. Although the usage of RDFLib facilitates the evaluation of subqueries and provides an efficient evaluation tool in each task, there is an overhead due to loading of each data segment, which is around 40 seconds for each data segment. Note that the loading time does not include the transfer time of each segment as well as the time that takes each task to be initialized.

We conducted several types of experiments to investigate both the performance of the query evaluation algorithms proposed in this paper and the impact of the query decomposition algorithms on the overall query evaluation. In the following, we initially analyze the scalability of each query evaluation algorithm, in terms of both the size of the dataset and the number of cluster nodes. Then, we analyze how the query evaluation algorithms perform in widely-used pattern types of queries and different partitions of the data graph (the ones mentioned above). Finally, we analyze how the overall performance of query evaluation is

¹RDFLib documentation: <https://rdflib.readthedocs.io/>

affected by the type of query decomposition selected.

8.1.1 Scalability of the Query Evaluation Algorithms

In this Section, we investigate the scalability of the QEJPE-algorithm, eval-STARS and QE-with-Redundancy algorithms. In particular, we conducted a set of experiments to analyze how the query evaluation algorithms perform in terms of both the size of the input dataset and the number of compute nodes in the cluster.

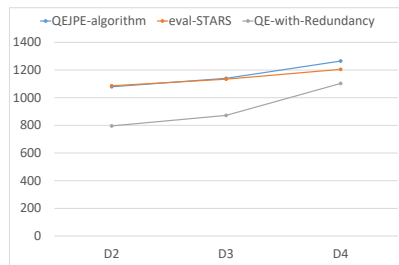
Initially, we selected three queries of different types from the WatDiv Benchmark and evaluated them using each of the algorithms over each of the D2-D4 datasets. The queries selected are illustrated in the Table 8.2, along with the number of subqueries generated per algorithm. For each query, the type of the query and the number of the resulting tuples for each dataset are included in the table, as well. Table 8.3 summarizes the execution time of each query, per evaluation algorithm and dataset, where L , S and F represent the Linear, Star and Snowflake queries selected, respectively. Figures 8.1a, 8.1b and 8.1c graphically show the execution time per dataset and evaluation algorithm, for each query. Figure 8.1d illustrates the average execution time for each dataset and each algorithm. Looking at the experimental results, we can see that although the amount of the data in each dataset is doubled (i.e., $D3$ and $D4$ have around 100% more triples than $D2$ and $D3$, respectively), the growth rate of the execution time remains less than 30%, in average; which shows that each algorithm scales well in terms of the size of the dataset².

Query type	Number of subqueries			Results		
	QEJPE-algorithm	eval-STARS	QE-with-Redundancy	D2	D3	D4
Linear (L2)	2	2	2	36	9	432
Star (S5)	2	1	1	20	33	64
Snowflake (F1)	3	2	2	0	10	4

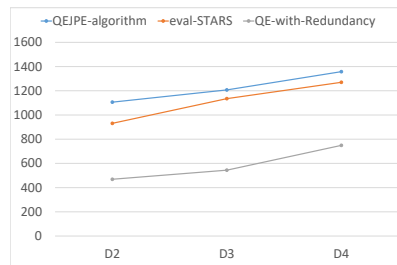
Table 8.2: Description of the queries evaluated over datasets of different sizes

To evaluate the scalability in terms of the size of the cluster (i.e., the number of compute nodes), we performed as follows. We evaluated over 3 cluster settings the 3 queries described in Table 8.2 using each algorithm over the dataset D4. In particular, the first setting had 4 compute nodes (NodeManagers), the second

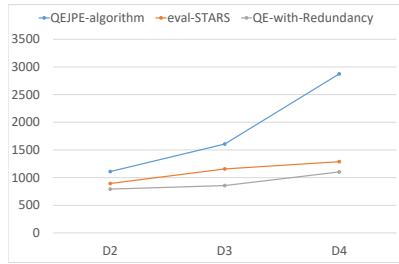
²Note that the scalability of each algorithm is limited by the capacity of the cluster resources (i.e., memory, disk space).



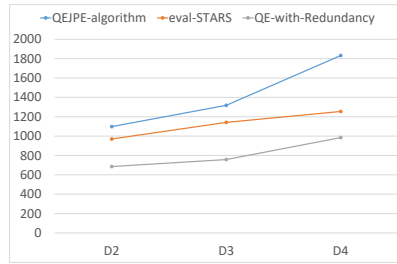
(a) Linear Query



(b) Star Query



(c) Snowflake Query



(d) Average per evaluation algorithm

Figure 8.1: Query evaluation in terms of the size of dataset (in seconds)

	QEJPE-algorithm			eval-STARS			QE-with-Redundancy		
	L	S	F	L	S	F	L	S	F
D2	1079	1106	1109	1086	931	893	796	469	792
D3	1140	1207	1606	1134	1135	1156	872	544	856
D4	1265	1358	2874	1205	1270	1288	1103	749	1102

Table 8.3: Query evaluation in datasets of different sizes (in seconds)

had 7 compute nodes and the last one utilized all the 10 available compute nodes. Then, we executed the evaluation algorithms in each setting. The execution times are summarized into the Table 8.4. Figures 8.2a, 8.2b and 8.2c illustrate the execution time in terms of the size of the cluster per algorithm for each type of query. As we can see, the algorithms scale well in terms of the number of compute nodes; i.e., the execution time is decreasing by increasing the number of compute nodes.

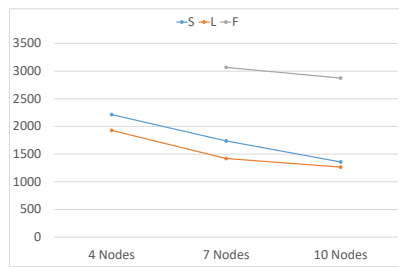
	QEJPE-algorithm			eval-STARS			QE-with-Redundancy		
	L	S	F	L	S	F	L	S	F
4 Nodes	1929	2214	-	1970	1914	2073	1957	1537	1993
7 Nodes	1421	1738	3967	1417	1415	1437	1323	969	1334
10 Nodes	1265	1358	2874	1205	1270	1288	1103	749	1102

Table 8.4: Query evaluation in D4 dataset for different compute nodes size (seconds)

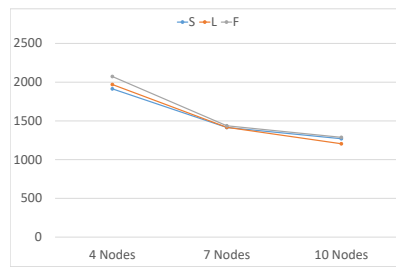
8.1.2 Comparison of Query Evaluation Algorithms

In this Section, we present the outcomes of the experiments performed in order to compare the performance of the query evaluation algorithms proposed in this work; i.e., the QEJPE-algorithm, eval-STARS and QE-with-Redundancy algorithms. The evaluation performed by applying all the algorithms for a variety of queries over the dataset D4 described in the previous Sections. We used multiple queries from WatDiv Benchmark, from all the proposed query types (Linear, Star, Snowflake, and Complex). Table 8.5 summarizes the queries used in this experiment, along with corresponding characteristics of each query; e.g., number of triples, number of variables, number of resulting tuples over the dataset D4, and the number of subqueries generated by the query decomposition.

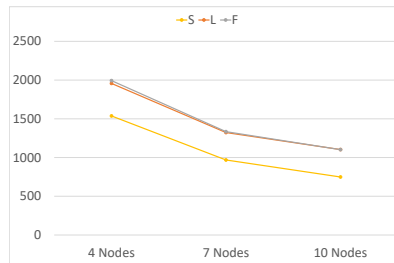
The QEJPE-algorithm and eval-STARS algorithms were also tested over two data partitioning approaches, random partitioning and METIS, while the QE-with-Redundancy algorithm was only evaluated over vertex-partitioned data (due to the requirements of the algorithm). The execution time for each query



(a) QEJPE-algorithm



(b) eval-STARS



(c) QE-with-Redundancy

Figure 8.2: Query evaluation in terms of compute nodes size per algorithm (seconds)

is included in the Table 8.6, where the execution time is given in minutes, followed by seconds (i.e., Minutes:Seconds). The average execution time for each algorithm and each query type, per data partitioning approach, is illustrated in Table 8.7 and graphically presented in Figure 8.3. Note that evaluating the majority of Snowflake and Complex queries using QEJPE-algorithm the cluster reached the memory limits (14.5GB for all YARN containers on a node) and did not manage to provide any result.

Parameters	Linear		Star		Snowflake		Complex
	L2	L4	S3	S5	F1	F4	C3
Triples	4	3	4	4	6	9	6
Variables	2	2	4	3	5	8	7
Results	432	109	677	64	4	71	763924
Subqueries QEJPE-algorithm	2	1	2	2	3	5	3
Subqueries eval-STARS and QE-with-Redundancy	2	1	1	1	2	2	1
Subqueries QE-with-Redundancy	2	1	1	1	2	2	1

Table 8.5: Description of Watdiv queries

Query	QEJPE		eval-STARS		QE-with-Redundancy
	METIS	Random	METIS	Random	
L2	21:06	21:05	21:11	20:05	18:23
L4	19:56	21:06	21:09	21:03	12:43
S3	21:47	34:59	18:51	21:18	12:37
S5	20:43	22:38	18:14	21:10	12:29
F1	23:29	47:54	22:22	21:28	18:22
F4	-	-	21:22	21:38	18:23
C3	-	-	30:09	25:30	17:48

Table 8.6: Execution time per query

Query Type	Data Partitioning	Linear	Star	Snowflake	Complex
QEJPE-algorithm	METIS	1231,0	1275,0	-	-
	Random	1265,5	1728,5	-	-
eval-STARS	METIS	1270,0	1112,5	1312,0	1809,0
	Random	1234,0	1274,0	1293,0	1530,0
QE-with-Redundancy	s-decomposition	933,0	753,0	1102,5	1068,0

Table 8.7: Average execution time per query type (seconds)

As we can see in the experimental results, QEJPE is more efficient for Linear queries than Star queries. In addition, METIS outperforms Random partition for both Star and Linear queries.

Queries L2 and L4 from WatDiv Benchmark (Linear query type) evaluated in all the algorithms and the mean execution times in seconds of these queries

are presented in Figure 8.3a and in Table 8.7. QE-with-Redundancy algorithm performs better than QEJPE-algorithm and eval-STARS algorithms. eval-STARS algorithm perform better than QEJPE-algorithm using both METIS and Random partition.

Queries S3 and S5 were used to evaluate star type queries. QE-with-Redundancy algorithm performs better than QEJPE-algorithm and eval-STARS algorithms while eval-STARS perform better than QEJPE-algorithm. Both eval-STARS and QEJPE-algorithm perform better for METIS partition than random partition.

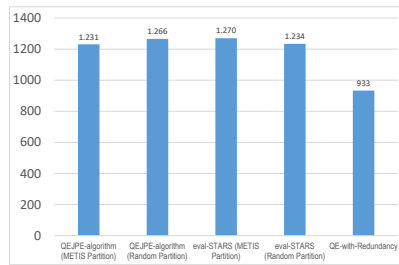
In case of Snowflake queries, queries F1 and F4 executed. Experimental results prove that QEJPE-algorithm is not efficient for this type of queries. QE-with-Redundancy algorithm performs again better results from eval-STARS algorithm. eval-STARS algorithm performed almost the same results for random and METIS partition.

Similar behavior with Snowflake queries had the Complex type queries. C3 query executed and QEJPE-algorithm was not efficient, QE-with-Redundancy algorithm performs better results than eval-STARS algorithm. In this type of query, eval-STARS algorithm performed better using random partition rather than METIS partition.

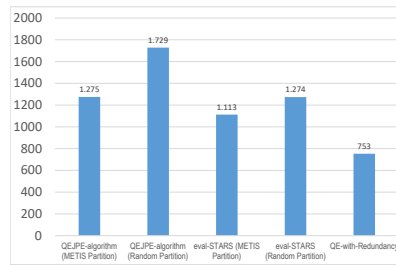
8.1.3 Query Decomposition Algorithms Evaluation

In this Section, we experimentally analyze how the selection of the query decomposition algorithm can affect the overall query evaluation performance. We focus on the three main query decomposition algorithms proposed in Section 6.3; i.e., min-res, max-degree, and max-degree-with-reshaping. To perform this experiment, we decomposed multiple queries using the aforementioned decomposition algorithms and evaluate them using a single evaluation algorithm and over a single dataset.

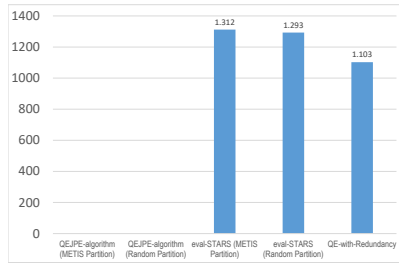
In particular, we initially used a query template (i.e., query graph structure) over the WatDiv data model and generated six different queries by setting variables and constants to the nodes. The queries Q1-Q6 that were constructed are depicted in Figure 8.4, where the white-colored nodes represent the variables and dark-colored nodes represent constants. We also constructed an additional complex query Q7, over the WatDiv data model, asking for certain edges of the data graph multiple times. We then decomposed the queries Q1-Q6 using different decomposition algorithms and evaluated them using the QE-with-Redundancy



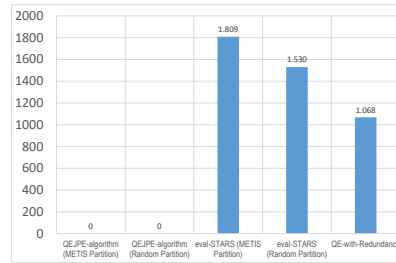
(a) Linear query type evaluation



(b) Star query type evaluation



(c) Snowflake query type evaluation



(d) Complex query type evaluation

Figure 8.3: Comparison of query evaluation algorithms for a variety of query types (seconds)

algorithm and the dataset D4. For Q7 query, the smaller dataset D1 was used to overcome memory limitation due to the large number of results. The execution time for each query and each decomposition algorithm is illustrated in Table 8.8, along with the number of subqueries resulted by each decomposition algorithm and the number of resulting tuples. The execution time per query and algorithm is graphically presented in Figure 8.5.

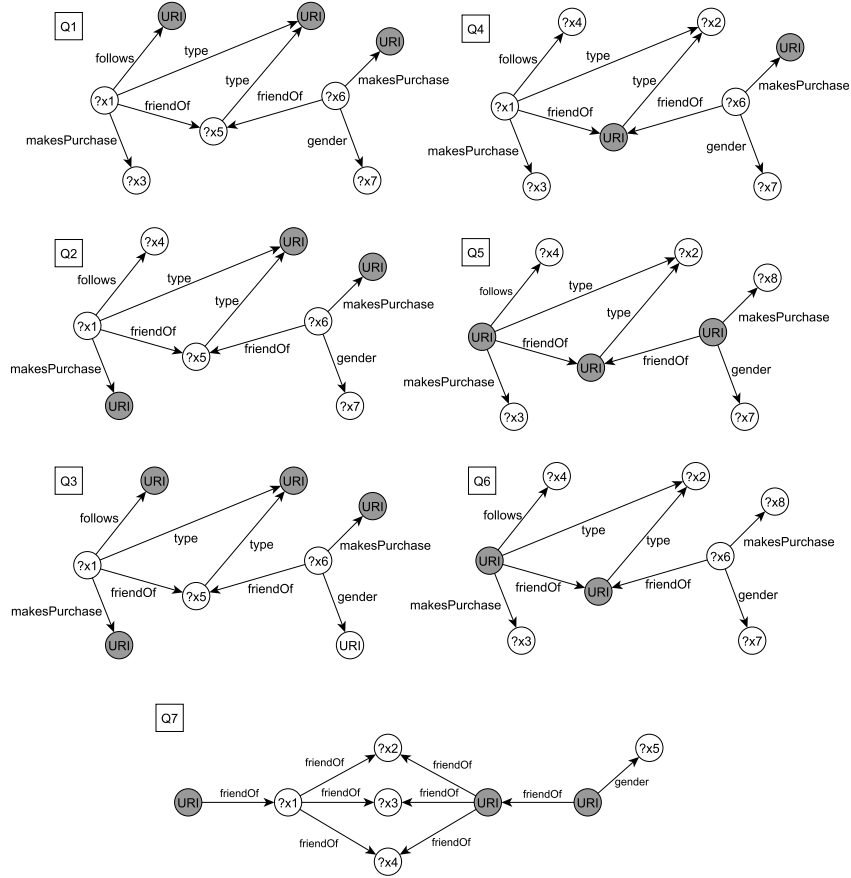


Figure 8.4: Queries used to compare decomposition algorithms

Analyzing the execution time of the queries per decomposition algorithm (Table 8.8 and Figure 8.5), we can easily see that for queries Q1 and Q2, the decompositions of the max-degree and max-degree-with-resaping perform better than the ones given by min-res. Queries Q4-Q6 perform similarly, for all the three algorithms. This can be explained by the fact that the queries Q1 and Q2

Queries	Results	min-res		max-degree		max-degree-reshaping	
		subqueries	time	subqueries	time	subqueries	time
Q1	11	5	1784	3	1103	3	1103
Q2	33	5	1544	3	1129	3	1129
Q3	1	3	1144	3	1144	3	1144
Q4	1580	5	1422	3	1430	3	1430
Q5	5808	6	1436	3	1431	3	1429
Q6	12705	6	1428	3	1432	3	1425
Q7	438976	7	1132	3	1305	3	1331

Table 8.8: Evaluation using QE-with-Redundancy and different query decompositions (Seconds)

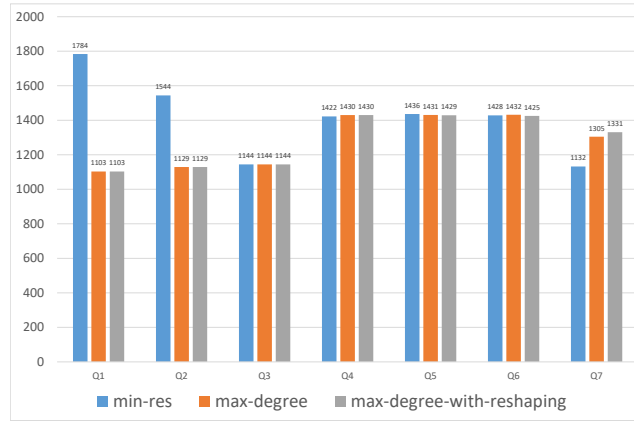


Figure 8.5: Query decomposition algorithms evaluation (seconds)

give decompositions with more subqueries using min-res than the decompositions given by the two other algorithms, as well as the results are few. For queries Q4-Q6 the performance of min-res algorithm is improved compared with the max-degree and max-degree-with-reshaping algorithms. Query Q3 is an exception since the decomposition resulted by all the three algorithms is the same.

Comparing now the execution time of the max-degree and the max-degree-with-reshaping, these algorithms resulted similar decompositions. Hence, as we can see in Table 8.8 and Figure 8.5, their execution time for the majority of the queries is very close.

Looking however the execution time of the query Q7, the decomposition

resulted by min-res outperforms the decompositions given by max-degree and max-degree-with-resaping. To analyze this result in more detail, we can easily see that since the node x_1 is high degree, gives a subquery with multiple variables in the decompositions given by max-degree and max-degree-with-resaping. In addition, the variables x_2 , x_3 , and x_4 in both subqueries map to the same data nodes and increase significantly the number of intermediate results (comparing with the number of the corresponding data edges mapped by these variables in all the embeddings). On the other hand, min-res handles such a case better, since it does not allow subqueries having more than 2 variables to be generated.

8.2 Evaluation of the Doc-based Algorithm

To evaluate the Doc-based Algorithm through experimentally analyzing the performance and scalability of the overall approach a set of experiments were conducted using Apache Spark and MongoDB. Spark was initially used for preparing and partitioning the input data graph, as well as loading the data segments into a distributed collection of a MongoDB Cluster (see Subsection 7.5.2). MongoDB cluster was used for storing the data segments and efficiently answering subqueries over the data segments. Finally, an application providing a query interface and managing the overall query evaluation process was implemented. The application takes a query from the user, decomposes the query into subqueries (see Section 7.3) and translates subqueries into MongoDB queries, as described in Subsection 7.5.3. The subqueries are then used to build the proper Spark job which is submitted into the Spark cluster (see Subsection 7.5.3) in order to get back the query answers.

In the experimental analysis we used 10 virtual machines (VMs) to setup the Spark (6-node cluster) and MongoDB (1 router server, 1 config server and 5 shards) clusters. The specifications of each VM are given as follows: Intel Xeon 2.30GHz (8 Cores) with 16GB RAM, 60GB HD, Ubuntu 16.04. In all the experiments, the amount of memory per executor process was set to 6GB, the number of executors was set to 20, and the executor cores was set to 3.

We used the Waterloo SPARQL Diversity Test Suite (WatDiv) to generate 4 different datasets D1-D4 (in N-Triples format). WatDiv generates benchmark datasets which are widely used (e.g., [113, 60]) to evaluate and measure the performance of RDF data management systems and query evaluation engines, since it provides a wide spectrum of SPARQL queries.

Table 8.9: Datasets characteristics

Dataset	Size (GB)	Triples	Segments
D1	1.16	8,773,357	417,705
D2	2.33	17,582,410	834,665
D3	3.50	26,342,929	1,251,625
D4	4.67	35,112,532	1,668,585

Table 8.9 describes the datasets; showing the size of each dataset, the number of triples generated and the number of segments stored in the MongoDB as documents.

Furthermore, we set up five indexes to each dataset’s collection (1 collection for each dataset) in order to improve the performance of finding the embeddings of subqueries over each segment. The selected indexes are the values of the keys `centralNode`, `subject.predicate.value`, `subject.predicate.object.value`, `object.predicate.value` and `object.predicate.subject.value`. The indexes were selected in such a way that if a subquery includes constants (i.e., either URIs or literals), these constants are taken into consideration in order to improve the performance of the subquery evaluation.

To evaluate our approach, we used 17 WatDiv benchmark queries, which cover a wide range of conventional query forms that are practically used in the real datasets. According to the WatDiv Benchmark, the queries are categorized into linear queries (L) (queries L1, L3, L4), star queries (S) (queries S1, S2, S3, S5, S6, S7), snowflake-shaped queries (F) (queries F1, F2, F3, F4, F5) and complex queries (C) (queries C1, C2, C3). Typically, linear refers to path queries, star category of the benchmark includes simple star queries (e.g., having at least one URI as a leaf), and snowflake refers to star consisting of star subqueries (i.e., star of stars). Complex category includes queries that could not categorized into one of the aforementioned categories, or queries of the aforementioned categories with high selectivity [120] (i.e., large amount of data in the result). Their query definitions are described, in details, in the WatDiv documentation (see *Query definitions/Templates*: <https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml>). The 17 queries selected are listed in the first column of Table 8.10.

Table 8.10 summarizes both the main information about the selected queries (such as the number of the generalized star subqueries $\mathcal{F}(Q)$, and the number

$|Q|$ of the query triples) and the evaluation results for each dataset (in seconds). Notice that the queries L3, L4, S1, S2, S3, S5, S6, S7, C3 are generalized star queries; hence, the number of created subqueries is 1. In addition, the queries L2, F1, F2, F3, F4, F5 are decomposed into 2 subqueries, while the query C1 is decomposed into 3 subqueries and the query C2 is decomposed into 5 subqueries. Note here that the number of subqueries implies the number of joins required during query execution (e.g., L2 requires a single join of the results of the subqueries in order to find the answer of L2, while L3 requires no join).

The outcomes of our experimental analysis are illustrated in the Figures 8.6, 8.7 and 8.8. Figure 8.6 graphically illustrates the execution results included in Table 8.10, for each query and each dataset, while Figure 8.7 illustrates the average execution time of queries per benchmark category, as well as the scaling ratio through the R^2 (*R-squared*) measure. The average execution time of queries per benchmark category are also summarized into Table 8.11. Analyzing further the scalability of our approach w.r.t. the dataset size, Figure 8.8 illustrates the scalability per number of subqueries. In Figure 8.8, each line represents the average, relative percentage (across all the datasets) of execution time of the queries decomposed into X subqueries, where X is the name of the line (i.e., 1, 2, 3, or 5 subqueries). The relative percentage of execution time is given by dividing each execution time of each query with the maximum execution time of the query across all datasets, and describes the percentage of execution time w.r.t. the maximum execution time of each query; e.g., the execution time L1 for D1 is 48.57% ($= \frac{8.65 \times 100\%}{\max(8.65, 12.60, 17.13, 17.81)}$) of the max execution time of L1 across all datasets (i.e., of the execution time of L1 over D4).

Figures 8.7 shows that, for each benchmark category of queries, our approach scales linearly w.r.t. the dataset size. Additionally, Figure 8.8 shows that our approach scales linearly (w.r.t. the dataset size) regardless the number of subqueries. On the other hand, Figure 8.6 shows that the number of subqueries (e.g., L4 vs. L1) and the size of the result (e.g., L2 vs. C3) significantly affect the execution time; the execution time increases w.r.t. the number of subqueries (i.e., joins required) and the size of the query answer.

Table 8.10: Execution times for 17 WatDiv benchmark queries (seconds).

Query Q	$ Q $	$ \mathcal{F}(Q) $	D1	D2	D3	D4
L1	3	2	8.65	12.60	17.13	17.81
L3	2	1	4.10	4.97	5.76	6.99
L4	2	1	3.06	3.95	4.95	5.65
S1	9	1	2.56	3.39	4.12	4.93
S2	4	1	3.09	4.30	5.77	6.52
S3	4	1	2.71	3.81	4.58	5.64
S5	4	1	2.53	4.17	5.18	6.55
S6	4	1	2.35	3.58	4.11	5.32
S7	3	1	1.96	3.04	3.69	4.74
F1	6	2	5.34	7.25	9.81	9.74
F2	8	2	7.64	11.25	17.79	21.60
F3	6	2	28.06	37.91	45.82	44.83
F4	9	2	4.16	5.64	7.46	12.05
F5	6	2	19.84	27.15	66.69	96.73
C1	8	3	18.08	32.85	61.82	76.09
C2	10	5	51.01	140.87	197.51	248.05
C3	6	1	8.30	13.29	35.76	31.92

Table 8.11: Average execution times of the queries grouped according to the benchmark query categorization (seconds)

Query Type	D1	D2	D3	D4
(L)inear	5.27	7.17	9.28	10.15
(S)tar	2.53	3.71	4.58	5.62
Snow(F)lake	13.01	17.84	29.51	36.99
(C)omplex	25.80	62.33	98.37	118.69

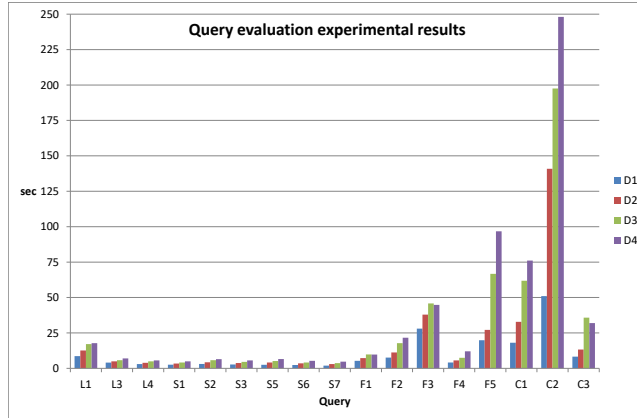


Figure 8.6: Query evaluation experimental results

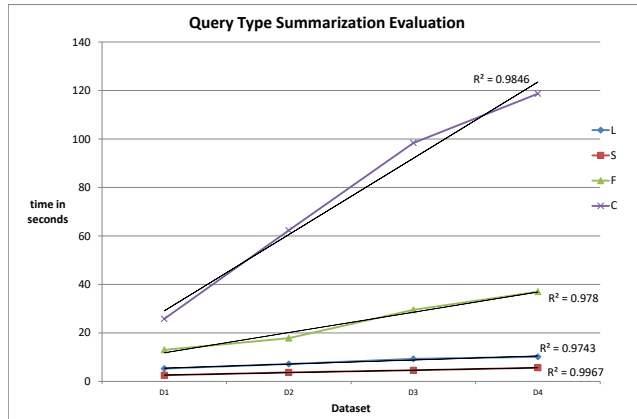


Figure 8.7: Query type summarization evaluation experimental results.

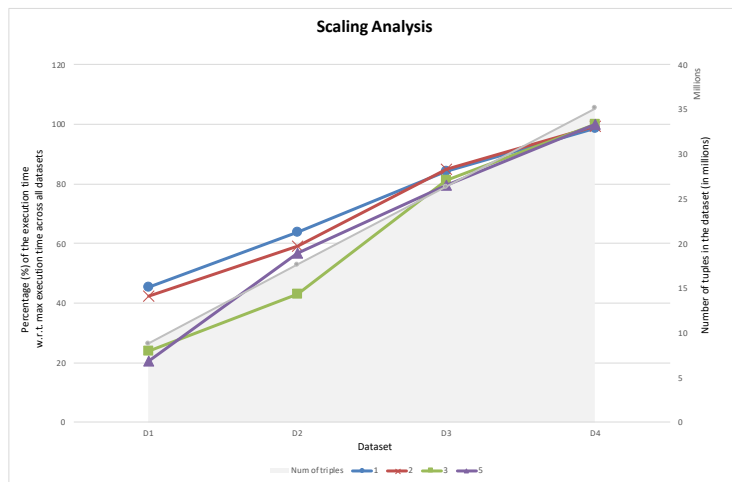


Figure 8.8: Scaling analysis.

Chapter 9

Related work

The problem of efficiently querying linked data have been widely investigated, in both centralized (e.g., [116, 117]) and distributed environments (e.g., [68, 107]). In addition to the query evaluation approaches, a variety of effective storage schemes have been used to improve query answering, such as the use of relational databases in centralized approaches (e.g., [42, 117]) and NoSQL databases (e.g., [128]) in distributed environments. To query linked data, we mainly focus on SPARQL as query language.

9.1 Centralized Environments for Querying Linked Data

In centralized environments, there is a variety of approaches used for storing linked data. Widely-used approaches include the *triple table* (e.g., [90, 92, 123]), the *property tables* (e.g., [124, 36]) and the *vertical partitioning* (e.g., [82, 51, 25]). In the triple-table approach each RDF triple is stored as a triple in a table with three columns; subject, predicate, and object. In property tables, all the triples sharing the same value of the subject are stored in a single record of a table, while in the vertical partitioning approach all the triples sharing the same predicate values are stored into the same two column table. Furthermore, storing the linked data into a normalized relational model (e.g., [117, 116]) has also been extensively investigated. In the majority of such approaches, the query evaluation is performed by translating SPARQL into SQL queries (e.g., [42, 117, 116]). In addition, certain indexing strategies are applied in order to optimize query

answering.

It's worth mentioning RDF-3X [90, 92] and Hexastore [123], two widely-used systems that use the triple-table approach. B+ tree indices for all the different combinations of the subject, predicate and object are applied, where the triples are sorted lexicographically in B+ tree. Additionally, these systems use indices with statistics and compressed methods for better performance.

Unlike using relational databases as storage layer, there have been investigated approaches utilizing graph structures to store linked data. *Graph-Based storage* (a.k.a., Vertex-Based, or Node-Based) methods focus on the storing RDF data into a directed, multi-edge graph. gStore [135] is a representative example, where the data graph is stored into disk-based adjacency-list table.

In the case of central processing methods vertical scaling has maximum limits and can not be used when the amount of data increases rapidly [41]. To handle the problem of querying large volumes of linked data, a variety of distributed methods for storing and processing linked data have been proposed [128]. Next sections focus on analyzing these approaches.

9.2 Using Distributed Storage and Processing

As mentioned earlier, approaches applied on a single machine lack scalability. To handle scalability on query evaluation, a variety of big data technologies [128] and NoSQL databases have been used. Most of the approaches proposed in the literature to handle scalability of answering SPARQL queries over big linked datasets [26] focus on two aspects, distributed storage of linked data and distributed processing of SPARQL queries. Typically, the proper partitioning of the data into a distributed repository (either file system or distributed NoSQL database) can significantly improve the performance of query answering [41, 51, 74]. Following this approach, most of the distributed based methods and systems utilize efficient partitioning of the data across a cluster of machines in order to ensure efficient query processing through minimizing the communication cost and improving parallel execution [41].

Since linked data is represented by a data graph [84], balanced graph partitioning is investigated in order to store linked data in a distributed manner. However, finding a balanced graph partition, as proved, is a NP-complete problem [54]; hence, the majority of the partitioning techniques proposed focus on either efficiently approximating a balanced partition using conventional graph

partitioning methodologies (e.g., [75]) or applying certain patterns that are implied by the form of linked data graph (e.g., semantic partitioning).

The most commonly used partitioning methods [41] are: *Random Partitioning*, *Hash Partitioning*, *Workload-aware Partitioning*, *Graph Partitioning* and *Semantic Partitioning*. Random Partitioning [56, 93] is the simplest partitioning method which divides the total triples of the data graph into separate parts by taking a portion of its number of edges. It is fast method and offers balanced files but it does not offer efficient processing features. The Hash Partitioning method [107, 49, 44] partitions the RDF triples according to their subject, predicate, object value, or a combination of them. In some cases, RDF triples are replicated across more than one partition to improve processing efficiency. The approaches that use the hash partitioning method can evaluate commonly used star queries on a local level, which minimizes the communication costs. Workload-aware Partitioning [52] is based on previous query workloads (query log files). As such, partitions are used to locally evaluate query types that have been frequently used by users in the past. By using the Graph Partitioning method [67], RDF data is partitioned according to their graph structure. Min-cut graph partitioning software METIS [75] is typically used in such cases. METIS reduces the number of graphs connecting nodes between different partitions but has scalability issues. This method has the main advantage of placing connected nodes in the same partition but does not fully guarantee it, so common queries like star queries cannot always be evaluated locally. An ontology is usually used to partition data according to their semantic structures within Semantic Partitioning [81]. In addition to their syntax, IRIs are also used as semantic partitioning method [81]; assuming that IRIs that have the same hierarchy prefix are often queried together.

To take advantage of the selected partitioning during query answering, certain approaches for decomposing the given query, and creating a proper query plan (consisting of multiple steps of distributed processing) are proposed [49, 58]. Most of these approaches [74, 45, 26] use a distributed processing framework, such as MapReduce [5] or Apache Spark [11], to execute the steps identified by the query plan. Apart from these approaches, there is a noteworthy amount of related work focusing on utilizing distributed NoSQL databases [104, 34, 110, 78] to store the linked data graph and answer the given queries. In these cases, the query evaluation is achieved either through translating the given SPARQL query into the query language supported by the NoSQL database used [87], or by using a distributed processing framework to implement the overall query execution

plan [58] (in such a case, the NoSQL database mainly used as a storage layer ensuring proper data partitioning).

In the following section, we analyze approaches investigated in the literature which propose distributed methodologies for evaluating SPARQL queries over a large amount of distributed linked data. These approaches are classified into three main categories, the ones using MapReduce/Hadoop (e.g., [107, 68]) as the main distributed processing layer, the ones using Apache Spark (e.g., [60, 113]) for this purpose, and a set of approaches which either propose a custom distributed processing framework (e.g., [107]) or are built on top of a NoSQL database (e.g., [98]). Then, we also analyze additional distributed evaluation approaches that cannot be classified into one of the aforementioned categories.

9.2.1 Query Answering Using MapReduce

In this section, we focus on analyzing approaches for evaluating SPARQL queries using the MaReduce paradigm. The main challenge of evaluating SPARQL queries over a large, distributed set of linked data is to apply certain joining operators over the data that is distributed on multiple machines. Afrati et al. [29] investigated this problem for relational data and proposed an approach for optimizing joins in MapReduce by choosing the appropriate map-key and shares in order to minimize the communication cost. This approach is extended in [28] to data graphs, where the cost of evaluating queries on data graphs using one round of Map-Reduce is investigated, and an approach of translating the query patterns into conjunctive queries is proposed. The communication cost is minimized using the techniques of the approach proposed in [30]. Such an approach could be used for answering conjunctive SPARQL queries [91, 103, 121].

A method of answering SPARQL Basic Graph Pattern using traditional multi-way join into MapReduce, instead of multiple individual joins, is also presented in [88], where certain joining keys are selected to avoid unnecessary iterations. This approach can be used for every type of partitioning of the RDF data.

SHARD [107] is built on top of Hadoop, and uses the Hadoop distributed files system (HDFS) to store data in native text files. The data is not indexed, or replicated, and no query plan is used to minimize the number of MapReduce jobs (or to reduce the intermediate results). It uses subject hash partitioning to decompose the RDF data graph; all the triples with the same subject are stored in the same line of the text file. For the execution, one MapReduce job

is created for every query triple, while an additional job is used, at the end, to remove duplicated results and apply the required projection. Hence, assuming an n -triple query pattern, $n + 1$ jobs are required, and all the data graph is scanned n times.

HadoopRDF [68] uses predicate hash partition method to distribute the data graph; similar to the vertical partitioning approach applied by SW-Store [25]. In general, the number of the data fragments is equal to the number of the distinct predicates. If, however, a predicate value belongs to multiple triples (e.g., `rdf:type` value) the subgraph is further partitioned into smaller blocks, according to the distinct object values. The data is stored into flat files in HDFS. The query evaluation then is performed through a sequence of MapReduce jobs. HadoopRDF query processing is optimized using a heuristic and a greedy approach which produces a query plan to minimize the number of MapReduce jobs and the intermediate results.

CliqueSquare [58] presents a method that generates highly parallelizable query plans for BGP queries, which rely on n -ary equality joins with minimum amount of MapReduce stages. CliqueSquare uses a data partitioning scheme that permits first-level joins can be evaluated locally at each node. The proposed partitioning method extends the default three replicas Hadoop fault-tolerant replication mechanism, since each data triple is partitioned according to its subject, property and object values. The triples that share the same value in subject, predicate or object are placed on the same node. This partition ensures that queries sharing the same variable (like star queries) can be evaluated locally. Additionally, on each node the data graph is vertically partitioned.

H2RDF [98] uses the Apache HBase [7] to store data triples. Three RDF indices on subject, predicate and object (spo, pos and osp combinations) are materialized and stored to HBase in the form of key-value pairs. Different strategies are used to execute joins and answer the given query. H2RDF+ [97] extends H2RDF by considering three more indices (ops, osp and sop). Furthermore, MapReduce Merge Join algorithm is used to join query triples that share the same variable and the MapReduce Sort-Merge Join algorithm is used for joining the intermediate results.

PigSPARQL [111] is yet another approach which uses Hadoop-based implementation of vertical partitioning of the data stored into HDFS. It implements a translation from SPARQL to Pig Latin [94] and uses Apache Pig [10] as an intermediate engine. SPARQL query triples are reordered for query optimization processing. In the system RAPID+ [77], an alternative query algebra, called the

Nested Triple Group Algebra, is used as an extension of Apache Pig, to improve the performance of SPARQL query processing over MapReduce.

The authors in [67] proposed a graph partitioning schema, which resembles the s-decomposition partitioning defined in this work. In particular, the data is partitioned in such a way that the vertices that are relatively close to each other are included in the same segment. In this context, the following main methodologies are investigated and proposed: the undirected and the directed n -hop guarantee. The former focuses on initially partitioning the vertices and then assigning the triple-paths of length n that start from a vertex that is already included in the segment. The latter is similar to the directed one but considers any undirected path of length n . In both cases, a graph practitioner tool which is based on METIS is used for partitioning the vertices of the RDF graph into disjoint partitions so that the minimum number of edges is cut. The queries are also decomposed in such a way that the subqueries generated can be computed locally, in each cluster node. MapReduce is used for the joins of the intermediate results of subqueries. Although s-decomposition partitioning approach is similar to 1-hop undirected guarantee (or hash partitioning), the n -hop guarantee of the data graph may cause data explosion especially in coherent data graphs if $n > 2$.

SHAPE [81] proposes a semantic hash partitioning which is based on the similarity of the URI hierarchy of the vertices. The vertices with same URI prefixes are placed in the same partition. After a simple hash partitioning is used, a replication of only a set of necessary triples is performed, using a k -hop semantic hash partitioning and context-aware filters. The system also uses a RDF-3X triple store in each data node. Query processing and the joins of the intermediate results is based in MapReduce.

The papers [71, 93, 56] focus on both decomposing queries and partition the RDF data, where the data is stored into MySQL and the framework used to evaluate the queries is MapReduce. SPARQL to SQL translations is used for query processing, and MapReduce is used to apply the joins. Note however, that all the suggested methods work independently the storage method (files or databases). Although these approaches ensure management (even for complex queries) of large amount of RDF data, the response time of the queries is questionable, due to the performance limitation of the MapReduce computational engine.

D-SPARQ [87] uses the document database MongoDB [17] to store and index data using subject hash partition. A single MapReduce job is then used to import data into the document database and to collect statistical information

for query optimization process based on join reordering. All triples sharing the same subject value are stored in the same document (JSON) file.

Another approach which is based on the MapReduce is Sempala [112], which applies SPARQL-to-SQL translation on top of Hadoop. It uses Impala [8] as a distributed SQL processing engine. Sempala uses a unified vertical partitioning (single property table) in order to boost the star-shaped queries.

In [122], the authors proposed a MapReduce algorithm, called StarMR, which is based on star decomposition for answering subgraph matching queries. The StarMR algorithm is improved with two optimization strategies. The first applies an RDF property filtering approach and the second one postpones any Cartesian product operation. RDF graph is stored in a distributed adjacency list.

[69] uses a partitioning method over the predicate value and the type of objects to store the RDF data. Query processing is performed using MapReduce and the algorithm proposed applies a number of the jobs that depends on the form of the given query.

9.2.2 Query Answering Using Spark

In this section, we focus on analyzing distributed query evaluation over RDF data, taking advantage of the Spark computation engine. Apache Spark is mainly used to improve the performance of SPARQL query evaluation over big RDF data [31].

SPARQLGX [60] uses a vertical partition approach, where the triples are partitioned according to their predicate values. The query evaluation is performed by initially filtering the triples matching a query triple, in each segment, and then, by applying a sequence of join operations through a query plan which is generated according to predefined statistics.

S2RDF [113] also proposes a vertical-like partitioning, called Extended Vertical Partitioning (ExtVP), which is based on semi-joins reductions (i.e., a certain number of semi-joins are applied between the vertical partitioning tables and their results are materialized for improving the overall performance). To evaluate queries over ExtVP, an approach of applying a certain partitioning of the query triples (in order to achieve parallel/local computation) and utilizing Spark SQL is followed.

HAQWA [44] proposes a hash-based partitioning over the subject values of the RDF triples. This ensures local computation of subject-centric star queries (a subclass of generalized star queries). To extend the supported queries, the query

is decomposed into subqueries and missing triples of each subquery are replicated. The overall computation process is managed through a Spark application.

The authors in [89] analyze the query evaluation plan of a BGP expression on Spark and proposes a joins plan for efficiently evaluating BGPs over a large RDF graph. Considering an initially hash-based partitioning of the data (e.g., the triples are partitioned by their subject), the authors propose a hybrid method to find a query plan. The approach uses a cost-driven combination of partitioned/cascade and broadcast joins over Spark.

Apart from the previous approaches, it's worth mentioning the approaches S2X [109], Spar(k)ql [59] and [76], which focus on evaluating SPARQL queries using the Spark GraphX library. Sparklify [119] applies a SPARQL-to-SQL rewriter for translating SPARQL queries into Spark executable code.

In [64], a property table scheme is built on top of HBase storage system and a vertical partitioning scheme on top of Cassandra storage system. Query processing is based on SPARQL query translation to SparkSQL for both HBase and Cassandra storage schemas.

9.2.3 Using NoSQL Databases

As mentioned in the previous sections, there are multiple approaches that use NoSQL platforms to store RDF data and answer SPARQL queries. In the context of MapReduce and Spark-oriented query evaluation approaches (see previous sections), we discussed several approaches utilizing such systems. In this section, we extend this analysis by analyzing similar approaches, where the query execution layer is more bound to the underlying systems used [74].

The distributed systems Rya [104], AMADA [34], MAPSIN [110] and CumulusRDF [78] use NoSQL [47] databases to store RDF data and provide efficient query processing using three different indices SPO, POS and OSP (S for subject, P for predicate and O for object values). More specifically, Rya use Apache Accumulo [2], AMADA use Amazon DynamoDB [1], MAPSIN Apache HBase and CumulusRDF Apache Cassandra [3]. MAPSIN (Map-Side Index Nested Loop Join) joins are performed in the map phase, so shuffle and reduce phase are not required. The proposed algorithm is optimized for the efficient processing of multiway joins.

9.2.4 Other Distributed Query Evaluation Engines

In this section, we present additional approaches that propose distributed evaluation frameworks that do not utilize any particular distributed processing engine.

EAGRE [132] uses a key-value storage layer to store the data in a distributed manner and an I/O efficient strategy for efficient SPARQL queries using specified solution sequence modifier evaluation. METIS is used in EAGRE to find a proper data partitioning scheme. In [101], the data graph is also partitioned using METIS tool but the approach is based on the partial evaluation and assembly framework [70] and not to the query decomposition into smaller subqueries. gStore-D [134] is a distributed RDF management system that adopt partial evaluation and assembly strategy.

The problem of finding an effective query decomposition approach has also been investigated in the context of multi-query optimization in federated RDF systems. Particularly, [99] proposes a cost-driven method to distributively evaluate queries in a federated system. The approach identifies the common subgraphs of an initially-given set of queries, and decomposes the queries into a set of subqueries, taking into account the data availability of each source. Then, using a source topology graph, a query plan is constructed and distributively evaluated.

Partout [52] uses a workload partitioning method based on query logs. The partition takes care properties which are queried together. RDF-3x is used for RDF storage and query processing in the nodes. In WARP [65], a workload-aware partitioning and replication method is used. This approach is based on the directed 2-hop replication partitioning method and extends it with data replication, taking into account frequent queries. The data is stored in RDF-3X, on each node, and query processing is performed by a dedicated processing layer which utilizes the data partitioning. SPARTI [83] uses the SemVP partitioning schema which extends the vertical partition with row-level semantics based on the join patterns found in the query workload. The query evaluation is also performed through a custom, dedicated processing layer which exploits the underlying partitioning approach. In [126], a path partitioning technique is proposed partitioning initial data into multiple path preserving data focus on complex SPARQL queries.

Query evaluation based on partial results is presented in [100]. The suggested method extends [101]. Partial results are compressed into a compact data

structure and then all the partial results are merged together to form the final results. The size of the query and the partitioning of the graph affects the communication cost of the partial results. Moreover, optimization based on the communication of the variables internal candidates is presented.

AdPart [32], is a distributed in-memory RDF engine. Subject hash partitioning strategy is used for data partitioning and the query processing takes advantage of the data locality in order to minimize the communication cost. Furthermore, AdPart monitors the query workload and incrementally redistributes parts of the data that are frequently accessed by frequent patterns. By maintaining and indexing these patterns, many future queries are evaluated without communication. The adaptivity feature of AdPart complements its excellent performance on queries that can benefit from its hash-based data locality. Frequent query patterns that are not favored by the initial partitioning (e.g., star joins on an object) can be processed in parallel due to adaptivity.

Adaptive Hashing (AdHash) [63] is another distributed in-memory RDF engine that initially distributes triples by hashing on their subjects for maximizing the number of joins that can be executed in parallel without data communication for the common star queries. Then data triples redistributed based on the workload.

In DiploCloud [127], semantically related data is stored together to minimize internode operations. Molecule clusters, template lists, and a molecule index are used as data structures. Molecule clusters are a type of property table, while template lists store literals in lists.

In JARS [105] distributed RDF storage system, data is hashed twice based on the subject and object. Relation database is used for data storage and query processing is based on SPARQL to SQL transformations. Each triple is stored twice, with all the triples that have the same subject or object, to be placed together on the same machine. The database in each machine contains two tables, one for the triples with the same subject and the other for the triples with the same object. Data partition provides the ability star and simple chain SPARQL queries to be executed locally.

DREAM [62] does not partition the data graph but stores the entire dataset on each cluster node. The initial query is decomposed into many subqueries based on a query planner and each subquery is executed into a separate node. RDF-3X is used for data storage on each node.

ScalaRDF [66] is another distributed and scalable RDF triple store based on in-memory key-value store. Data placement and elastic resource scale out/in

is based on a protocol that extends the consistent hashing protocol. ScalaRDF supports updating operations.

Trinity.RDF [131], is a distributed, memory-based graph engine based on Trinity [22], a distributed in-memory key-value store, that stores RDF data in its native directed graph form and supports more advanced graph analytic on RDF data. Two different key value pairs data model are used to store nodes, according to the number of their neighbors. In case the nodes have few neighbors, a random partitioning is used which is based on the sharding that is supported by the underlying key-value store. Otherwise, the partitioning is optimized to overcome the network communication cost. Query processing is treated a sequence of patterns instead of matching single patterns independently to reduce the join cost.

It's worth mentioning an approach for transforming SPARQL queries into Apache Flink [4] programs for querying RDF data [40]. Apart from the translation approach, the authors in [40] provides a set of formal definitions of Apache Flink's subset transformations. The semantic correspondence between Apache Flink's subset transformations and the SPARQL Algebra operators is also presented.

Chapter 10

Conclusions

Since the amount of linked data is rapidly increasing, the efficient querying of large amount of data becomes a significant challenge. In this Ph.D. thesis, we proposed a family of algorithms for querying large amount of linked data in a distributed environment, such as the World Wide Web.

The general idea behind these algorithms is as follows:

- the data graph G is decomposed into a set of (possibly overlapping) data graph segments stored in different nodes of a cluster of commodity computers.
- the query graph Q is also decomposed into a set of (possibly overlapping) subqueries.
- Subqueries are applied to each data graph segment, in isolation, and intermediate results are computed.
- The intermediate results are appropriately combined to obtain the answers of the query Q .

At first we proposed a general algorithm called QEJPE-algorithm which computes partial embeddings of subqueries of a query Q over data segments of a decomposed data graph G and combining these partial embeddings to obtain (total) embeddings of the initial query Q in the data graph G . QEJPE-algorithm computes the answers to a given query correctly, independently of a) the data graph decomposition, b) how graph segments are stored, c) the query decomposition, and d) the algorithm used for calculating (partial) results.

Then we developed the algorithms called eval-STARS and QE-with-Redundancy algorithms which are based on a) query decompositions which produce subqueries of special forms (with or without replication of query triples) (eval-STARS and QE-with-Redundancy algorithms) and b) decompositions of the data graphs in which replication is allowed in data graph segments in such a way that answers to a the special form of queries, can be obtained from a single data segment. (QE-with-Redundancy algorithm)

The proposed algorithms are independent on the way the data are stored as well as of the particular implementation of the query evaluation. Finally, we proposed a specific implementation of the proposed algorithms, using MapReduce framework.

Extended experimental evaluation exhibited that all algorithms scale well in terms of a) the number of compute nodes; i.e., the execution time is decreasing by increasing the number of compute nodes and b) the size of the data graph; i.e., the amount of the data in each dataset is doubled, the growth rate of the execution time remains less than 100%.

The last approach is based on the idea that RDF data can be efficiently stored on a document database through partitioning the overall data-graph into star segments. The proposed data partitioning approach ensures high-performance query answering for a wide sub-class of queries, called generalized star queries. We also proposed a document-oriented data model for storing effectively the data graph in a distributed document database in order to utilize the distributed query processing mechanism of such systems. MongoDB is used as a representative NoSQL database for implementing our approach. The class of generalized star queries extends the conventional star queries, which are extensively investigated in the related literature, so that both subject-object and object-subject edges are allowed in the queries. The efficiency of the approach is based on the feature that no joining operations are required between different data collections in order to completely evaluate a generalized star query over each data record/segment, and the segments could be evenly distributed in multiple cluster nodes. Our approach is then extended to support the evaluation of general BGP SPARQL queries. The evaluation of these queries is based on the fact that every BGP query can be decomposed into a set of generalized star queries which can be evaluated independently of each other (and in parallel), and then the results of this evaluation can be appropriately combined to get the answers of the initial BGP query. Experimental evaluation shows also that our last approach is scalable in terms of the size of the data graph and efficient in the query

evaluation.

Bibliography

- [1] Amazon DynamoDB. <https://aws.amazon.com/dynamodb/>.
- [2] Apache Accumulo. <https://accumulo.apache.org/>.
- [3] Apache Cassandra. <http://cassandra.apache.org/>.
- [4] Apache Flink. <https://flink.apache.org/>.
- [5] Apache Hadoop. <https://hadoop.apache.org/>.
- [6] Apache Hadoop YARN. <https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html>.
- [7] Apache HBase. <https://hbase.apache.org/>.
- [8] Apache Impala. <https://impala.apache.org/>.
- [9] Apache Jena. <https://jena.apache.org/>.
- [10] Apache Pig. <https://pig.apache.org/>.
- [11] Apache Spark. <https://spark.apache.org/>.
- [12] Apache TinkerPop. <https://tinkerpop.apache.org/>.
- [13] Couchbase. <https://www.couchbase.com/>.
- [14] Hadoop and MongoDB. <https://www.mongodb.com/hadoop-and-mongodb>.
- [15] JSON (JavaScript Object Notation). <http://www.json.org/>.
- [16] Memcached - A Distributed Memory Object Caching System. <https://memcached.org/>.

- [17] MongoDB, NoSQL Document Database. <https://www.mongodb.com/>.
- [18] Neo4j Graph Platform. <https://neo4j.com/>.
- [19] Optimal implementation of conjunctive queries in relational databases.
- [20] Redis. <https://redis.io/>.
- [21] Riak. <https://riak.com/>.
- [22] Trinity. <https://www.microsoft.com/en-us/research/project/trinity/>.
- [23] Virtuoso Universal Server. <https://virtuoso.openlinksw.com/>.
- [24] Waterloo SPARQL Diversity Test Suite (WatDiv). <https://dsg.uwaterloo.ca/watdiv/>.
- [25] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, 18(2):385–406, 2009.
- [26] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. A Survey and Experimental Comparison of Distributed SPARQL Engines for very large RDF Data. *Proceedings of the VLDB Endowment*, 10(13):2049–2060, 2017.
- [27] Foto N. Afrati and Rada Chirkova. *Answering Queries Using Views, Second Edition*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.
- [28] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. Enumerating Subgraph Instances Using Map-Reduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 62–73, April 2013.
- [29] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *Proceedings of the 13th International Conference on Extending Database Technology, EDBT ’10*, pages 99–110, New York, NY, USA, 2010. ACM.
- [30] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, Sept 2011.

- [31] Giannis Agathangelos, Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. RDF Query Answering Using Apache Spark: Review and Assessment. In *34th IEEE International Conference on Data Engineering, ICDE Workshops*, pages 54–59. IEEE Computer Society, 2018.
- [32] Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning.
- [33] Güneş Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified Stress Testing of RDF Data Management Systems. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig Knoblock, Denny Vrandečić, Paul Groth, Natasha Noy, Krzysztof Janowicz, and Carole Goble, editors, *The Semantic Web – ISWC 2014*, pages 197–212, Cham, 2014. Springer International Publishing.
- [34] Andrés Aranda-Andújar, Francesca Bugiotti, Jesús Camacho-Rodríguez, Dario Colazzo, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. AMADA: Web Data Repositories in the Amazon Cloud. In *21st ACM International Conference on Information and Knowledge Management, CIKM’12, Maui, HI, USA, October 29 - November 02, 2012*, pages 2749–2751, 2012.
- [35] Tim Berners-Lee. *Linked Data - Design Issues*, volume 2009. ACM Press, 2006. Publication Title: W3C.
- [36] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Ian Horrocks and James Hendler, editors, *The Semantic Web — ISWC 2002*, pages 54–68, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- [37] Gavin Carothers. RDF 1.1 N-Quads. W3C Recommendation, February 2014.
- [38] Gavin Carothers and Eric Prud’hommeaux. RDF 1.1 Turtle. W3C Recommendation, February 2014.
- [39] Gavin Carothers and Andy Seaborne. RDF 1.1 N-Triples. W3C Recommendation, February 2014.

- [40] Oscar Ceballos, Carlos Alberto Ramírez Restrepo, María Constanza Pabón, Andres M. Castillo, and Oscar Corcho. SPARQL2Flink: Evaluation of SPARQL Queries on Apache Flink. *Applied Sciences*, 11(15), 2021.
- [41] Tanvi Chawla, Girdhari Singh, Emmanuel S. Pilli, and M.C. Govil. Storage, Partitioning, Indexing and Retrieval in Big RDF Frameworks: A Survey. *Computer Science Review*, 38:100309, 2020.
- [42] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics Preserving SPARQL-to-SQL Translation. *Data & Knowledge Engineering*, 68(10):973–1000, 2009.
- [43] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O’Reilly, 2010.
- [44] Olivier Curé, Hubert Naacke, Mohamed-Amine Baazizi, and Bernd Amann. HAQWA: a Hash-based and Query Workload Aware Distributed RDF Store. In *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015.*, CEUR Workshop Proceedings, vol. 1486, 2015.
- [45] Alfredo Cuzzocrea, Rajkumar Buyya, Vincenzo Passanisi, and Giovanni Pilato. MapReduce-based Algorithms for Managing Big RDF Graphs: State-of-the-Art Analysis, Paradigms, and Future Directions. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*, pages 898–905. IEEE Computer Society / ACM, 2017.
- [46] Ian Davis, Arnaud Le Hors, and Thomas Steiner. RDF 1.1 JSON Alternate Serialization (RDF/JSON). W3C Note, November 2013.
- [47] Ali Davoudian, Liu Chen, and Mengchi Liu. A Survey on NoSQL Stores. *ACM Computing Surveys*, 51(2):40:1–40:43, 2018.
- [48] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [49] Jin-Hang Du, Hao-Fen Wang, Yuan Ni, and Yong Yu. *HadoopRDF: A Scalable Semantic Data Analytical Engine*, pages 633–641. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

- [50] Schahram Dustdar and Ben Falchuk. Semantic Web. In Borko Furht, editor, *Encyclopedia of Multimedia*, pages 803–810. Springer US, Boston, MA, 2006.
- [51] David C. Faye, Olivier Curé, and Guillaume Blin. A Survey of RDF Storage Approaches. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, 15:11–35, 2012.
- [52] Luis Galárraga, Katja Hose, and Ralf Schenkel. Partout: a Distributed Engine for Efficient RDF Processing. In Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel, editors, *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 267–268. ACM, 2014.
- [53] Fabien Gandon and Guus Schreiber. RDF 1.1 XML Syntax. W3C Recommendation, February 2014.
- [54] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-complete Problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.
- [55] Lars George. *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly, 2011.
- [56] Manolis Gergatsoulis, Christos Nomikos, Eleftherios Kalogeros, and Matthew Damigos. An Algorithm for Querying Linked Data Using Map-Reduce. In A. Hameurlain, J. W. Rahayu, and D. Taniar, editors, *Data Management in Cloud, Grid and P2P Systems - 6th International Conference, Globe 2013, Prague, Czech Republic, August 28-29, 2013. Proceedings*, volume 8059 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2013.
- [57] Seth Gilbert and Nancy Lynch. Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.
- [58] François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. CliqueSquare: Flat Plans for Massively Parallel RDF Queries. In *International Conference on Data Engineering*, Seoul, South Korea, 2015.

- [59] Gergő Gombos, Gábor Rácz, and Attila Kiss. Spar(k)ql: SPARQL Evaluation Method on Spark GraphX. In M. Younas, I. Awan, and J. E. Haddad, editors, *4th IEEE International Conference on Future Internet of Things and Cloud Workshops, FiCloud Workshops 2016, Vienna, Austria, August 22-24, 2016*, pages 188–193. IEEE Computer Society, 2016.
- [60] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. SPARQLGX: efficient distributed evaluation of SPARQL with Apache Spark. In P. T. Groth et al., editors, *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II*, volume 9982 of *Lecture Notes in Computer Science*, pages 80–87. Springer, 2016.
- [61] Claudio Gutierrez, Carlos A. Hurtado, Alberto O. Mendelzon, and Jorge Pérez. Foundations of Semantic Web Databases. *Journal of Computer and System Sciences*, 77(3):520–541, 2011.
- [62] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti, and Sherif Sakr. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *Proceedings of the VLDB Endowment*, 8(6):654–665, 2015.
- [63] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, and Nikos Mamoulis. Evaluating SPARQL Queries on Massive RDF Datasets. *Proceedings of the VLDB Endowment*, 8(12):1848–1851, 2015.
- [64] Mahmudul Hassan and Srividya K. Bansal. Semantic Data Querying over NoSQL Databases with Apache Spark. In *2018 IEEE International Conference on Information Reuse and Integration, IRI 2018, Salt Lake City, UT, USA, July 6-9, 2018*, pages 364–371. IEEE, 2018.
- [65] Katja Hose and Ralf Schenkel. WARP: Workload-Aware Replication and Partitioning for RDF. In Chee Yong Chan, Jiaheng Lu, Kjetil Nørsvåg, and Egemen Tanin, editors, *Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1–6. IEEE Computer Society, 2013.
- [66] Chunming Hu, Xixu Wang, Renyu Yang, and Tianyu Wo. ScalaRDF: A Distributed, Elastic and Scalable In-Memory RDF Triple Store. In *22nd IEEE International Conference on Parallel and Distributed Systems*,

- ICPADS 2016, Wuhan, China, December 13-16, 2016*, pages 593–601. IEEE Computer Society, 2016.
- [67] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.
 - [68] Mohammad Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1312–1327, 2011.
 - [69] Mohammad Farhan Husain, Latifur Khan, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. In *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*, pages 1–10. IEEE Computer Society, 2010.
 - [70] Neil D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.
 - [71] Eleftherios Kalogeros, Manolis Gergatsoulis, and Matthew Damigos. Redundancy in Linked Data Partitioning for Efficient Query Evaluation. In I. Awan, M. Younas, and M. Mecella, editors, *3rd International Conference on Future Internet of Things and Cloud, FiCloud 2015, Rome, Italy, August 24-26, 2015*, pages 497–504. IEEE Computer Society, 2015.
 - [72] Eleftherios Kalogeros, Manolis Gergatsoulis, and Matthew Damigos. Document Based RDF Storage Method for Efficient Parallel Query Processing. In *Metadata and Semantic Research (MTSR 2018)*, volume 846 of *Communications in Computer and Information Science*, pages 13–25, Limassol, Cyprus, 2019. Springer, Cham.
 - [73] Eleftherios Kalogeros, Manolis Gergatsoulis, and Matthew Damigos. Document-Based RDF Storage Method for Parallel Evaluation of Basic Graph Pattern Queries. *International Journal of Metadata, Semantics and Ontologies*, 14(1):63–80, 2020.
 - [74] Zoi Kaoudi and Ioana Manolescu. RDF in the Clouds: A Survey. *The VLDB Journal*, 24(1):67–91, 2015.

- [75] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
- [76] Besat Kassaie. SPARQL over GraphX. *CoRR*, abs/1701.03091, 2017.
- [77] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. From Sparql to MapReduce: The Journey Using a Nested Triplegroup Algebra. *Proceedings of the VLDB Endowment*, 4(12):1426–1429, 2011.
- [78] Günter Ladwig and Andreas Harth. CumulusRDF: Linked Data Management on Nested Key-Value Stores. In *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, volume 30, 2011.
- [79] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *Operating Systems Review*, 44(2):35–40, 2010.
- [80] Doug Laney. 3D Data Management: Controlling Data Volume, Velocity and Variety. *META Group Research Note*, 6(70), 2001.
- [81] Kisung Lee and Ling Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *Proceedings of the VLDB Endowment*, 6(14):1894–1905, September 2013.
- [82] Yongming Luo, François Picalausa, George H. L. Fletcher, Jan Hidders, and Stijn Vansummeren. *Storing and Indexing Massive RDF Datasets*, pages 31–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [83] Amgad Madkour, Walid G. Aref, and Ahmed M. Aly. SPARTI: Scalable RDF Data Management Using Query-Centric Semantic Partitioning. In Sven Groppe and Le Gruenwald, editors, *Proceedings of the International Workshop on Semantic Big Data, SBD SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 1:1–1:6. ACM, 2018.
- [84] Frank Manola, Eric Miller, and Brian McBride. RDF Primer. *W3C Recommendation*, 10(1-107):6, 2004.
- [85] L. Masinter, T. Berners-Lee, and R. T. Fielding. Uniform Resource Identifier (URI): Generic Syntax. 2005.

- [86] Deborah McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, 2004. <https://www.w3.org/TR/2004/REC-owl-features-20040210/>.
- [87] Raghava Mutharaju, Sherif Sakr, Alessandra Sala, and Pascal Hitzler. D-SPARQ: Distributed, Scalable and Efficient RDF Query Engine. In E. Blomqvist and T. Groza, editors, *Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013*, volume 1035 of *CEUR Workshop Proceedings*, pages 261–264, 2013.
- [88] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. SPARQL Basic Graph Pattern Processing with Iterative MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud, MDAC '10*, pages 6:1–6:6, New York, NY, USA, 2010. ACM.
- [89] Hubert Naacke, Bernd Amann, and Olivier O. Curé. SPARQL Graph Pattern Processing with Apache Spark. In P. A. Boncz and Josep-Lluís Larriba-Pey, editors, *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, pages 1:1–1:7. ACM, 2017.
- [90] Thomas Neumann and Gerhard Weikum. RDF-3X: A RISC-Style Engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.
- [91] Thomas Neumann and Gerhard Weikum. Scalable Join Processing on very Large RDF Graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 627–640, 2009.
- [92] Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal*, 19(1):91–113, February 2010.
- [93] Christos Nomikos, Manolis Gergatsoulis, Eleftherios Kalogeros, and Matthew Damigos. A Map-Reduce Algorithm for Querying Linked Data Based on Query Decomposition Into Stars. In K. Selçuk Candan et al., editors, *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014*, volume 1133 of *CEUR Workshop Proceedings*, pages 224–231, 2014.
- [94] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data

- Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.
- [95] M. Tamer Özsu. A Survey of RDF Data Management Systems. *Frontiers of Computer Science*, 10(3):418–432, 2016.
 - [96] Jean Paoli, Tim Bray, Michael Sperberg-McQueen, François Yergeau, and Eve Maler. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, November 2008. <https://www.w3.org/TR/2008/REC-xml-20081126/>.
 - [97] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. H2RDF+: High-performance Distributed Joins Over Large-scale RDF Graphs. In *2013 IEEE International Conference on Big Data*, pages 255–263, Oct 2013.
 - [98] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2RDF: Adaptive Query Processing on RDF Data in the Cloud. In *WWW '12 Companion: Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 397–400, New York, NY, USA, 2012. ACM.
 - [99] Peng Peng, Qi Ge, Lei Zou, M Tamer Özsu, Zhiwei Xu, and Dongyan Zhao. Optimizing Multi-Query Evaluation in Federated RDF Systems. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1692–1707, 2019.
 - [100] Peng Peng, Lei Zou, and Runyu Guan. Accelerating Partial Evaluation in Distributed SPARQL Query Evaluation. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 112–123. IEEE, 2019.
 - [101] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. Processing SPARQL Queries over Distributed RDF Graphs. *The VLDB Journal*, 25(2):243–268, 2016.
 - [102] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In David et. al. Hutchison, editor, *The Semantic Web - ISWC 2006*, volume 4273, pages 30–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. Series Title: Lecture Notes in Computer Science.

- [103] François Picalausa and Stijn Vansummeren. What are Real SPARQL Queries Like? In *Proceedings of the International Workshop on Semantic Web Information Management*, pages 1–6, 2011.
- [104] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: A Scalable RDF Triple Store for the Clouds. In *1st International Workshop on Cloud Intelligence (Colocated with VLDB 2012), Cloud-I '12, Istanbul, Turkey, August 31, 2012*. ACM, 2012.
- [105] Anjali Rajith, Shoji Nishimura, and Haruo Yokota. JARS: Join-Aware Distributed RDF Storage. In Evan Desai, Bipin C. Desai, Motomichi Toyama, and Jorge Bernardino, editors, *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, Montreal, QC, Canada, July 11-13, 2016*, pages 264–271. ACM, 2016.
- [106] Guha Ramanathan and Dan Brickley. RDF Schema 1.1. W3C Recommendation, February 2014.
- [107] Kurt Rohloff and Richard E. Schantz. Clause-iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-store. In *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing, DIDC '11*, pages 35–44, New York, NY, USA, 2011. ACM.
- [108] S. Sakr and G. Al-Naymat. Relational Processing of RDF Queries: A Survey. *SIGMOD Record*, 38(4):23–28, 2010.
- [109] Alexander Schätzle, Martin Przyjacił-Zablocki, Thorsten Berberich, and Georg Lausen. S2X: Graph-Parallel Querying of RDF with GraphX. In F. Wang et al., editors, *Biomedical Data Management and Graph Online Querying - VLDB 2015 Workshops, Big-O(Q) and DMAH, Waikoloa, HI, USA, August 31 - September 4, 2015, Revised Selected Papers*, volume 9579 of *Lecture Notes in Computer Science*, pages 155–168. Springer, 2016.
- [110] Alexander Schätzle, Martin Przyjacił-Zablocki, Christopher Dorner, Thomas Hornung, and Georg Lausen. Cascading Map-Side Joins over HBase for Scalable Join Processing. In Achille Fokoue, Thorsten Liebig, Eric L. Goodman, Jesse Weaver, Jacopo Urbani, and David Mizell, editors, *Proceedings of the Joint Workshop on Scalable and High-Performance Semantic Web Systems, Boston, USA, November 11, 2012*, volume 943 of *CEUR Workshop Proceedings*, pages 59–74. CEUR-WS.org, 2012.

- [111] Alexander Schätzle, Martin Przyjaciół-Zablocki, and Georg Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proceedings of the International Workshop on Semantic Web Information Management, SWIM '11*, pages 4:1–4:8, New York, NY, USA, 2011. ACM.
- [112] Alexander Schätzle, Martin Przyjaciół-Zablocki, Antony Neu, and Georg Lausen. Sempala: Interactive SPARQL Query Processing on Hadoop. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandečić, Paul Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, volume 8796 of *Lecture Notes in Computer Science*, pages 164–179. Springer, 2014.
- [113] Alexander Schätzle, Martin Przyjaciół-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF Querying with SPARQL on Spark. *Proceedings of the VLDB Endowment*, 9(10):804–815, 2016.
- [114] Andy Seaborne and Gavin Carothers. RDF 1.1 TriG. W3C Recommendation, February 2014.
- [115] Andy Seaborne and Eric Prud'hommeaux. SPARQL Query Language for RDF. W3C Recommendation, January 2008.
- [116] Juan F. Sequeda and Daniel P. Miranker. Ultrawrap: SPARQL Execution on Relational Data. *Journal of Web Semantics*, 22:19–39, 2013.
- [117] Dimitrios-Emmanuel Spanos, Periklis Stavrou, and Nikolas Mitrou. Bringing Relational Databases Into the Semantic Web: A Survey. *Semantic Web*, 3(2):169–209, 2012.
- [118] Manu Sporny and Digital Bazaar. RDFa Lite 1.1. W3C Recommendation, March 2015.
- [119] Claus Stadler, Gezim Sejdiu, Damien Graux, and Jens Lehmann. Sparklify: A scalable software component for efficient evaluation of sparql queries over distributed rdf datasets.
- [120] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In J. Huai et al., editors, *Proceedings of the 17th*

International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008, pages 595–604. ACM, 2008.

- [121] Maria-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In *Extended Semantic Web Conference*, pages 228–242. Springer, 2010.
- [122] Xin Wang, Lele Chai, Qiang Xu, Yajun Yang, Jianxin Li, Junhu Wang, and Yunpeng Chai. Efficient Subgraph Matching on Large RDF Graphs Using MapReduce. *Data Science and Engineering*, 4(1):24–43, 2019.
- [123] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.
- [124] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proceedings of the First International Conference on Semantic Web and Databases*, pages 120–139. Citeseer, 2003.
- [125] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Ling Liu, and Hai Jin. Scalable SPARQL Querying Using Path Partitioning. In J. Gehrke et al., editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 795–806. IEEE Computer Society, 2015.
- [126] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Ling Liu, and Hai Jin. Scalable SPARQL Querying Using Path Partitioning. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 795–806. IEEE Computer Society, 2015.
- [127] Marcin Wylot and Philippe Cudré-Mauroux. DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud. *IEEE Transactions on Knowledge and Data Engineering*, 28(3):659–674, 2016.
- [128] Ibrar Yaqoob, Ibrahim Hashem, Abdullah Gani, Salimah Mokhtar, Ejaz Ahmed, Nor B. Anuar, and Athanasios V. Vasilakos. Big Data: From

Beginning to Future. *International Journal of Information Management*, 36(6):1231–1247, 2016.

- [129] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In S. D. Gribble and D. Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.
- [130] Mattei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In E. M. Nahum and D. Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’10, Boston, MA, USA, June 22, 2010*, 2010.
- [131] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A Distributed Graph Engine for Web Scale RDF Data. *Proceedings of the VLDB Endowment*, 6(4):265–276, 2013.
- [132] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 565–576, 2013.
- [133] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gStore: Answering SPARQL .Queries via Subgraph Matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, 2011.
- [134] Lei Zou and M. Tamer Özsu. Graph-Based RDF Data Management. *Data Science and Engineering*, 2(1):56–70, 2017.
- [135] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. GStore: A Graph-Based SPARQL Query Engine. *The VLDB Journal*, 23(4):565–590, 2014.

Abbreviations

W3C	World Wide Web Consortium
WWW	World Wide Web
SW	Semantic Web
XML	Extensible Markup Language
RDF	Resource Description Framework
RDFS	RDF Schema
OWL	Web Ontology Language
RIF	Rule Interchange Format
JSON	JavaScript Object Notation
URI	Uniform Resource Identifier
IRI	Internationalized Resource Identifier
URL	Uniform Resource Locators
HTTP	Hypertext Transfer Protocol
BGP	Basic Graph Pattern
DAG	Directed Acyclic Graph
LOD	Linked Open Data