Effective Partitioning of RDF Data for Distributed Query Answering



Fredah Banda

Magdalen College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2021

Acknowledgements

To my mentor and supervisor, Professor Boris Motik, you have been nothing short of amazing and without you this thesis would not have been possible. I want to thank you for your patient support, inquisitive nature and insightful feedback. You have pushed me to sharpen my thinking and brought my work to a higher level. You met me at my lowest point in my Oxford journey and you nursed me back to academic and mental health. I couldn't ask for more. Above all else, thank you for being human, for being kind to me and genuinely being there. Thank you so much!

I would also like to thank the Rhodes Trust for supporting me from the beginning. Special mention goes to Mary Eaton, Sheila Partridge, Charles Conn, Elizabeth Kiss, and Doyin Atewologun who have been amazing humans and have seen me as one too and went the extra mile to see me to completion. For an absolutely luxurious Oxford experience, flexibility and financial support, I thank members of Magdalen College especially Mark Pobjoy and the SSF team for keeping me and my baby girl alive when times were hard, Sarah Church who always accommodated me on my irregular schedule, Suzy Wood and the finance office for being flexible and kind when I was late on payments, and Liz Howdill in the Graduate Tutoring Office for processing my countless forms with urgency and lightening speed. Special thanks to my favourite people in college - the Magdalen College Porters especially Rob and John-Joe; thanks to whom, I'm now an expert rower and punter. Julie Sheppard from the Department of Computer Science who gave me options when I felt stuck. They have all made my stay at Oxford memorable. I thank you for supporting me.

In addition, I would like to thank my family and friends for their undying support. My mother Emely, you have been and will always be my hero. You are the true love of my life. This is for you. Antwine, I will never forget the sacrifices you made, the tears, the joy, everything. Papias for the late nights full of stress relieving laughter and silence when I needed it. Shawn for being there when things fell apart and for not judging me despite you being perfect. Finally, I could not have completed this thesis without the inspiration, joy, countless hugs, prayers and kisses from my daughter Zhanga. I love you. To my Father, Mr Papias Banda, who set in me the hunger for knowledge and made sacrifices for me to get an education but died before I could make it; **I will always love you** and there's nothing I want more than to have you here with me.

There's an African proverb, "It takes a village to raise a child." I am not here by just wit and good luck. I have never been an island and hopefully I'll never be. I have tons of amazing family and friends and if it wasn't for these people, I really don't know how I would have made it to where I am today and through this DPhil (PhD). In no particular order, I'm sincerely grateful to Mukolo Sikazwe, Favour Mandanji Nyikosa, Maka Bunda Tounkara, Nelisa Nchimunya Tebeka, Mary Page Jiyani, Boipelo Muchweene Mauchaza, Shmona Simpson, Lesego Letlape, Karen Mumba, Tonny Brian Muthee, Mwitwa Choonya, Milka Warwa Siaji, Deshon Jahrel, Christina Konuche, Katrina Malanga, Natasha Chilundika, Natasha Phiri, Ndjodi Ndeunyema, Kunda Chinku, Kopano Matlwa Mabaso, Chileshe Tobo Mukenani, Kunda Mutalima, Nachilala Munkombwe, Chichi and Muntanga Mudenda, Patience Magosha, Abigail Nachilongo Ruth Kaneka with her beautiful daughters Leah and Reene, Arnold Kaluba, Debra Chokwenda, Thomas De Toledo, Mayumbo Nyirenda, David Makadani Zulu, Monde Kalumbilo-Kabemba, Abraham Daniel Ngoyi, Himatambo Ngandu, Keith Mbata, Bonface Lisuba, Angela Mayembe, Sharon Mayembe, Naomi Tivomelenge Zyambo, Faith Banda, Mangani Banda, Chanda Chisunka Banda, Emmanuel Joshua, Kholiwe Tembo, Sylvia Tembo Kalobwe, Anthony Kalobwe, Sydney Mwape, Katongo Chileshe, Christabel Sakala, Milensu Kapaipi, Martha Lukopesha, Chilombo and Kanda Musa, Kalombo Chilongoshi, Billy Zimba, Michael Mwezi Munukayumbwa, Brighton Phiri, Tinovimbanashe Mhlanga, William Milimo Moonga, Patson Tembo, Chiboni Musweu, Chanda Kaliwile, Diana Mwale Lukopesha, Saxon Lukopesha, Mirriam Mwale, Temitope Ajileye, Theresa Malonga, Samson Chibuta, Bwalya Sunkutu, Daniel Shaka, Kabwe Mulenga, Sishe Nkonde, Alexander Khunga, Madalitso Chlikwela, Lweendo Mapani, Flora Kalaba Chitalu, Malama Kasanda, Stella Mwandu, Susan Mwila, Faith Kashweka, Roydah Chama, Zipphorah Chibuye, Blexce team, and Salifya Mugode. Sometimes you don't know how much a friend needs a good laugh, a hug, a smile but you supported me through this journey, sometimes unbeknownst to you. Thank you for spending your time with me and making it all worthwhile. I love you guys! We did it!! We finally did it!!!

Abstract

The growing popularity of Resource Description Framework (RDF) as a mode for data exchange and integration has resulted in the increased growth of RDF datasets. Some large scale RDF datasets cannot be stored and processed efficiently on a single node. A common approach to processing large RDF datasets is to partition the data in a cluster of shared-nothing servers and use a distributed query evaluation algorithm. It is commonly assumed in the literature that the performance of query processing in such systems is limited mainly by network communication. In this thesis, we show that this assumption does not always hold and we argue that more important than minimizing network communication, we should prioritise even workload distribution among servers when partitioning. Moreover, we present a new RDF partitioning method based on Louvain community detection, which drastically reduces communication, but without a corresponding decrease in query running times. This is because strongly connected partitions can lead to workload imbalance among the servers. We present a further refinement of our technique that aims to strike a balance between reducing communication and spreading processing more evenly, and our empirical evaluation shows that such an approach can improve load balance and hence reduce both communication and query times.

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Resource Description Framework 1
1.2 SPARQL 3
1.3 Data Partitioning 5
1.4 Query Answering 6
1.5 Contributions 8

2 Preliminaries **11**

3 Partitioning in Distributed Architectures **15**
3.1 Hash Partitioning 16
3.2 METIS Min-Cut Graph Partitioning 17
3.3 n-hop: Vertex Partitioning with Duplication 17
3.4 Weighted vertex partitioning with pruning 18
3.5 Triple Grouping 19
3.6 Rooted Subgraphs 20
3.7 Conclusion 21

4 Distributed Query Answering **23**
4.1 Data Exchange Operator 23
4.2 Existing RDF Systems 25
4.2.1 Systems Using Key-Value Stores 26
4.2.2 Systems Using Centralized RDF Stores 27
4.2.3 Systems Using Distributed RDF File Systems 30
4.2.4 Discussion 33
4.3 Conclusion 34

5 Community-Based RDF Graph Partitioning 35
5.1 Louvain Algorithm for Community Detection . . . 36
5.2 Rationale for Partitioning by Communities . . . 39
5.3 Our Partitioning Algorithms for RDF . . . 40
5.3.1 Pruning Phase . . . 40
5.3.2 Community Detection . . . 42
5.3.3 Community Allocation . . . 44
5.3.4 Partition Building . . . 49
5.3.5 Handling Incremental Dataset Updates . . . 50
5.4 Conclusion . . . 51

6 Experimental Setting 53
6.1 Test Datasets and Queries. . . . 53
6.1.1 WatDiv . . . 53
6.1.2 LUBM . . . 54
6.2 Test Systems. . . . 55
6.2.1 Hardware . . . 55
6.2.2 Evaluation Protocol . . . 56
6.3 Conclusion . . . 57

7 Query Times and Communication 59
7.1 Experiment Results . . . 59
7.2 Discussion . . . 60
7.3 Conclusion . . . 63

8 Server Workload Analysis 65
8.1 Experiment Results . . . 65
8.2 Discussion . . . 66
8.3 Conclusion . . . 73

9 Impact of Network Speed 75
9.1 Experimental Setup . . . 75
9.2 Evaluation Protocol . . . 76
9.3 Experiment Results . . . 76
9.4 Discussion . . . 77
9.5 Conclusion . . . 78

10 Conclusion and Outlook 79
10.1 Conclusion . . . 79
10.2 Future Work . . . 82

Appendices

A Queries Used in Evaluation **85**
A.1 LUBM Queries 85
A.2 Watdiv Queries 88

References **95**

List of Figures

1.1 RDF triple represented as a graph edge 3
1.2 Sample of a partial RDF graph 3

5.1 Grey edges to be pruned from sample graph 40
5.2 G' - Pruned sample graph 42
5.3 Strong communities extracted from G' 43
5.4 Small communities extracted from G' 44
5.5 Tight allocation of communities from G' 45
5.6 Loose allocation of communities from G' 48

List of Tables

7.1 LUBM query evaluation times (s) 60
7.2 WatDiv query evaluation times (s) 60
7.3 The number of partial binding messages sent on LUBM 61
7.4 The number of partial binding messages sent on WatDiv 61

8.1 LUBM partial bindings messages sent per server per query 67
8.2 WatDiv partial bindings messages sent per server per query 68
8.3 WatDiv query L4 answers produced per server 69
8.4 WatDiv query L4 matches made per server 69
8.5 Atom matches for WatDiv 71
8.6 Atom matches for LUBM in millions (M) 71

9.1 LUBM query times (s) with network delay (ms) 76

1

Introduction

The Semantic Web is a well defined and structured machine readable extension of the current web. It is designed to ensure the information on the web is in a standard format and the information it contains is unambiguous and well defined and thus easy for machines to read and understand. The information on the Semantic Web in not only comprised of resources, but relationships between the resources and is referred to as Linked Data. Resource Description Framework (RDF) [1] is the World Wide Consortium (W3C) recommended format for representing information on the Web. This ensures that Linked Data is represented in common standards, formats and technologies. This allows for easy processing, use and integration of Linked Data.

1.1 Resource Description Framework

Resource Description Framework (RDF) [1] is a widely-used data model for publishing data on the Web. It is used to make data exchange and integration on the web

easier. It is flexible as it has no schema, this allows the data to take any structure. This flexibility allows for complex relationships to be easily represented in RDF. The core element of RDF is a triple which comprises a subject, predicate and object and takes the form $\langle subject, predicate, object \rangle$. Each triple connects a single resource to another resource; the resource in the object position. A resource can be used to represent any object from a person to an article or something abstract and these resources can be connected to other resources. A resource in RDF is represented by either an Internationalized Resource Identifier (IRI) which is a unique identifier for the resource or a literal which is used to allocate a value such as an integer, a string, or a date. In the event that incomplete or unknown data needs to be represented, a blank node can be used. Blank nodes are particularly useful for representing relationships between the data without explicitly naming the resource.

An RDF triple can be represented as a predicate-labeled directed edge from the subject to the object as shown in Figure 1.1. The subject and object are represented as vertices where as the predicate forms the edge that connects the vertices. As a result, an RDF dataset comprised of triples can be represented as a graph and is often called an RDF graph. A sample partial RDF graph relating to a university *Univ1* can be seen in Figure 1.2. From the resources, we see a range of different resources from departments such as *Dept1* and people such as *Prof1*, classes such as *ub:Department* and relationships such as *ub:worksFor*. This graph is built from triples that describe the relationships between the graph resources. For example, the triple $\langle Student2, ub:name, Anne \rangle$ says that Student2 has the name Anne.

Unlike traditional databases, an RDF graph can contain both schema information and non-schema triples. An example can be seen from the RDF graph in Figure 1.2, it has an edge that can be represented as a triple $\langle ub:gradStudent, rdfs:subclassOf, ub:student \rangle$ which describes a subclass declaration that *ub:gradStudent* is a subclass of *ub:student*. This type of information is called schema information. Several frequently occurring concepts have been standardised by the W3C through RDF Schema (RDFS) [2]. Common concepts in RDF have been standardised by the W3C using RDF Schema (RDFS) [3]. RDFS extends RDF and facilitates the use

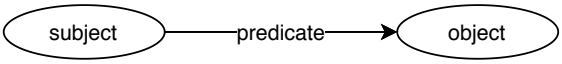


Figure 1.1: RDF triple represented as a graph edge.

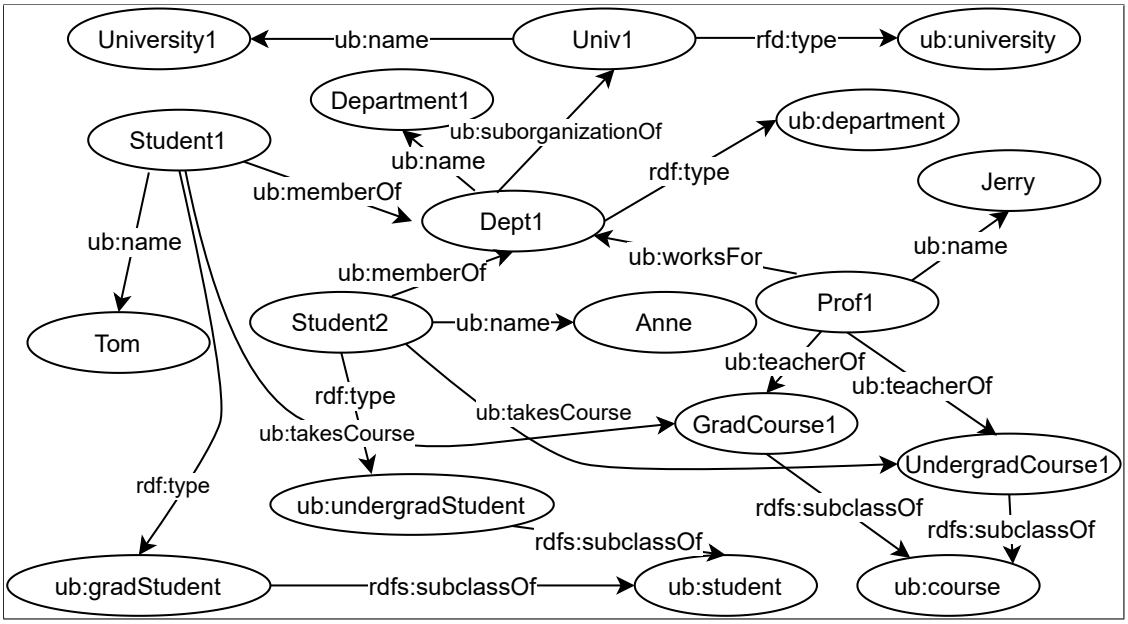


Figure 1.2: Sample of a partial RDF graph.

of standard vocabularies used across several RDF datasets. It comprises properties, classes and semantics that are interpreted in a standardized manner.

1.2 SPARQL

A good data representation technique requires a matching query language. SPARQL 1.1 [4] is the W3C standard language for querying RDF. In order to understand the semantics of SPARQL graph pattern expressions, we need to understand the concept of a partial mapping μ. A partial mapping μ from V to U is a partial function of the form $\mu : V \rightarrow U$. For a triple t of the form $\langle subject, predicate, object \rangle$, the triple obtained by replacing the variables in t according to μ is denoted by $\mu(t)$. It follows that the domain of μ represented as $dom(\mu)$ is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are compatible when for all $?X \in dom(\mu_1) \cap dom(\mu_2)$, it is the case that $\mu_1(?X) = \mu_2(?X)$, that is, when $\mu_1 \cup \mu_2$ is also a mapping. Two

mappings μ_1 and μ_2 are said to be compatible if μ_1 can be extended with μ_2 to obtain a new mapping and the μ_2 can also be extended by μ_1 [5].

The basic unit of a SPARQL query is a *basic graph pattern*, it follows the same format as a triple pattern of the form $\langle subject, predicate, object \rangle$ where one or more of the resources may be replaced with a variable. A SPARQL query containing only basic graph patterns is similar to a Select-Project-Join query in SQL. It comprises of two parts; the SELECT clause that is used to identify the variables that we would like in the results, and the WHERE clause that contains one or more *basic graph patterns*. Since a SPARQL query contains a conjunction of one or more basic graph patterns, they are sometimes referred to as conjunctive RDF queries. Such SPARQL queries follow the syntax shown in Equation 1.1 where var_i is a variable that represents a resource that occurs in one or more of the basic graph patterns b_j.

$$SELECT\ ?var_1 \ldots ?var_n\ WHERE\ \{b_1 \ldots b_m\} \tag{1.1}$$

Two basic graph patterns in a SPARQL query are said to be joined to each other if they share a common variable or resource. For example, suppose we are interested in querying the RDF graph in Figure 1.2. Below we have a SPARQL query to find the names of students that belong to the department Dept1 as well as the type of students they are. All the basic graph patterns in the query are joined by a repeating common variable ?*student* and is thus called a *star* query.

```
SELECT ?name, ?type WHERE {
        ?student ub:memberOf Dept1.
        ?student ub:name ?name.
        ?student rdf:type ?type }
```

The basic graph patterns in a query are sometimes referred to as atoms. The first atom will match ?*student* to *Student*1 and *Student*2. For each of these matches, we have to run the second and third atoms where the variable ?*student* is

replaced by *Student*1 and *Student*2. The final result of this query execution will be as shown below.

name	type
Tom	ub:gradStudent
Anne	ub:undergradStudent

While the example above is a simple SPARQL query, SPARQL is versatile query language and features functionalities common to several database query languages such as FILTER, LIMIT, UNION, ORDER BY, and OPTIONAL.

1.3 Data Partitioning

Resource Description Framework (RDF) is a widely-used data model for publishing data on the Web. Applications typically rely on RDF systems to store and answer SPARQL queries over RDF data; Wylot et al. [6] have recently surveyed the state of the art in RDF systems. Centralised systems such as Jena [7], Sesame [8], RDFSuite [9], and RDF-3X [10] store and process the entire dataset on a single server. However, datasets containing billions of triples are routinely encountered in practice, which often exceed the capacity of a single server.

A common solution is to partition a dataset across a cluster of shared-nothing servers and use a distributed query processing strategy. Distributing the dataset adds additional challenges to data storage, not only in the data structures used, but especially on which server to store the data as these decisions could affect the performance during query answering. Many distributed RDF systems have been developed, such as YARS2 [11], 4store [12], H-RDF-3X [13], Trinity.RDF [14], SHARD [15], SHAPE [16], Partout [17], AdPart [18], TriAD [19], SemStore [20], DREAM [21], and WARP [22] to name a few.

The choice of a partitioning strategy for a dataset is important as partitioning algorithms could significantly impact several aspects during query answering such as workload balance, query response time, communication overhead and storage

overhead. In addition to these aspects, different partitioning strategies require different resources, others may be memory intense and require a lot of time to generate the partitions like the Min-Cut graph partitioning algorithm from METIS [23] with others significantly faster and lightweight like hash partitioning which uses a hash function to decide to which partition the triple should be allocated. The choice of partitioning strategy thus depends on the aspects you want to optimise and the amount of time and resources available for the entire process.

The main goal of partitioning is to distribute the dataset in a manner that will aid efficient query answering. Since it is commonly assumed in the literature that network communication is the main bottleneck in distributed systems during query evaluation, it is the factor to minimize or optimize the most during partitioning. For some, this means reducing communication between servers. To address this, some partitioning strategies duplicate commonly used data across several servers, this is done to reduce the need for distributed join processing during query evaluation. While introduction of duplicate triples could improve performance, the improvement comes at the expense of introducing storage overhead. Some approaches use the introduction of duplicate triples to guarantee partition locality for queries of a certain path length. Other approaches guarantee the locality of certain types of queries, for example, if the partitioning method assigns triples by *subject*, then we know that any star query, that is a query where all atoms share a common subject, can be evaluated locally without the need for distributed join processing.

1.4 Query Answering

SPARQL as a query language is declarative, and as such, how the query is evaluated is completely dependent on the database management system on which the dataset resides. The database management system upon receiving a query, generates a query evaluation plan that is followed to generate the results. A query evaluation plan is an ordered set of operations required to produce query answers. A good query evaluation plan could minimise query execution costs by reducing evaluation

time and the use of resources. For example, computing a join on two sets of data, one indexed by a resource X of size 1 and the other indexed by a resource Y of size 10. You could scan X first, then use an index to scan Y which will result in one lookup, or you could scan Y first, then use an index to scan on X which would result in 10 lookups which is significantly more.

Besides the query execution plan, the mode of execution is also cardinal to the overall performance of query answering. For example, if you had a cluster of 10 shared nothing servers and the triples that could match an atom are all located on a single server; a database management system that broadcasts every atom in the query to all the servers in the cluster will cause matches to be searched for on all the servers, even those that may not contain possible matches. This is unnecessary lookups which could be avoided. Another approach would be to send the atom only to the servers in the cluster that could contain potential matches, which in this case is a single server.

One of the main challenges of distributed RDF systems is distributed join processing. Abdelaziz et al. [24] surveyed 22 and evaluated 11 distributed RDF stores on a variety of data and query loads, and they showed AdPart [18] and TriAD [19] to be the best performing. Join processing is the main technical problem in these systems: all triples participating in a join need to be transmitted over the network to a common server. Since network communication is much slower than accessing local disk or RAM, it is commonly assumed that reducing or even eliminating communication is key to the scalability of distributed RDF stores. Thus, systems usually aim to colocate groups of related triples and thus decrease communication cost. Some systems also duplicate data—that is, they place the same triple on several servers; while this can completely eliminate communication, it can also significantly increase storage requirements [13].

Recently, Potter et al. [25] presented a query answering approach based on *dynamic data exchange*. This approach uses an index to locate triples participating in a join so it can be used with any data partitioning strategy; moreover, it aims to avoid communication whenever joining triples are colocated, which leads to

competitive query answering times. In most systems, data partitioning is coupled with a query answering algorithm, which uses knowledge about how the data is partitioned to locate join counterparts. Our partitioning approach aims at reducing communication between servers during query evaluation, and as such, the approach by Potter et al. [25] that eliminates unnecessary communication is most suitable.

1.5 Contributions

In this thesis we present novel techniques and algorithms in the domain of partitioning of RDF for distributed RDF systems, as well as extensive research into the effects of partitioning that may have been previously overlooked in the literature. We summarise those contributions below.

In Chapter 5 we present two versions of our novel community graph partitioning algorithm for RDF. It is based on Louvain community detection [26]—a well-known algorithm for detecting groups of highly connected vertices in a graph. By assigning strong communities to servers as a whole, we significantly minimise the need for distributed join processing during query evaluation hence reduced communication which is often a bottleneck in distributed systems. Our approach aims to maximise local computation of answers on individual servers whilst avoiding high storage overhead and without the introduction of duplicate triples. Since a query is basically a map of one or more subgraphs from the main RDF graph, if strong communities are allocated unbroken to servers, when a query is evaluated, the chance that the entire query can be matched on a server without the need for distributed join processing is higher. Interestingly, while our approach is effective in reducing the amount of communication, the relationship between communication and query evaluation time is not as straightforward as was previously assumed in the literature. The limitations of our approach are clearly outlined in our empirical evaluation.

In Chapters 7, 8, and 9, we look at the effects of partitioning on query times, workload balance, as well as network speed. We evaluated our technique on the LUBM [27] and WatDiv [28] benchmarks against two well-known data partitioning

techniques: subject hash partitioning (which assigns each triple to the server obtained by hashing the subject value), and the weighted vertex partitioning with pruning by Potter et al. [25] (see Section 3). To our surprise, the relationship between network communication and query answering performance turned out to be more complex than commonly assumed. Our techniques are very effective in reducing network communication (by orders of magnitude in some cases), but this did not translate into a reduction in query answering times; in fact, our technique was sometimes worst-performing. A careful analysis revealed this to be due to the workload imbalance between the servers.

When servers contain strongly interconnected communities, query answers are often produced by data on a handful of servers; thus, only some servers are involved in query processing, while others are largely idle. In contrast, with subject hash partitioning, the data contributing to query answers is likely to be evenly distributed across the cluster, and so query processing tends to be evenly distributed as well. Since the overall amount of work is the same in both cases and server processing is parallelised, subject hash partitioning can be more efficient than community partitioning, even though the latter involves significantly less communication. Motivated by these observations, we further refined our *community partitioning - tight* partitioning technique to produce *community partitioning - loose* which is also presented in Chapter 5. For the loose coupling community partitioning, we aim to strike a balance between producing strongly connected communities and distributing the server workload evenly in the cluster. Our further experiments show that, on certain queries, such an approach can reduce network communication as well as improve query answering times.

2

Preliminaries

The Resource Description Framework (RDF) data model is commonly used for publishing structured data on the Web. RDF data is constructed using *resources*, which can be *IRIs*, *blank nodes*, or *literals* (e.g., strings or integers). An *RDF triple* is an expression of the form $\langle s, p, o \rangle$, where s, p, and o are resources called *subject*, *predicate*, and *object*, respectively. An *RDF graph* G is a finite set of triples. SPARQL is the standard language for querying RDF graphs. In this thesis, we consider only the fragment of SPARQL covering *conjunctive queries* (CQs)—that is, queries of the form

$$\mathsf{SELECT}\ ?X_1 \ldots ?X_n\ \mathsf{WHERE}\ \{\ s_1\ p_1\ o_1\ .\ \ \ldots\ \ s_n\ p_n\ o_n\ \},$$

where each s_i, p_i, and o_i is a resource or a variable. Each $s_i\ p_i\ o_i$ is called a *triple pattern*. Conjunctive queries correspond to the select–project–join fragment of relational algebra, and they form the basis of SPARQL.

We study the problem of answering CQs over an RDF graph G stored in a cluster consisting of ℓ shared-nothing, connected servers. For convenience, we identify each server with a unique integer k between 1 and ℓ. We assume that the RDF graph is partitioned into ℓ *partition elements* $P_1, \ldots, P_\ell$—that is, each P_k is the subset of G stored at server k, and $\bigcup_{1 \leq k \leq \ell} P_k = G$. We assume that no data is duplicated, so $P_k \cap P_{k'} = \emptyset$ holds for all $1 \leq k < k' \leq \ell$.

To locate triples participating in a join, a distributed query evaluation typically uses knowledge about how data is partitioned. In this thesis, we evaluate conjunctive SPARQL queries using the distributed query answering technique by Potter et al. [25], which aims to avoid communication when triples participating in a join reside on the same server. We next briefly summarise the main aspects of this algorithm on an example of evaluating a conjunctive query over an RDF graph split into partition elements $P_1 = \{\langle a, r, b\rangle, \langle b, s, c\rangle\}$ and $P_2 = \{\langle b, s, d\rangle\}$.

$$\mathsf{SELECT}\ ?X_1\ ?X_2\ ?X_3\ \mathsf{WHERE}\ \{\ ?X_1\ r\ ?X_2\ .\ ?X_2\ s\ ?X_3\ \}$$

The query is evaluated using index nested loop joins over a left-deep query evaluation plan. A left-deep query evaluation plan is one where all the internal nodes in the join processing tree are join operators and have at least one leaf as a child, the leaves are base relations, while the edges show the flow of data [29]. The query is sent to all servers, and each server starts asynchronously matching the first triple pattern of the query. In our example, server 2 cannot match the first triple pattern in P_2; in contrast, by matching the first triple pattern to triple $\langle a, r, b\rangle$, server 1 produces a *partial binding* μ that maps variables $?X_1$ and $?X_2$ to resources a and b, respectively. The second triple pattern can now be matched both in P_1 and P_2. To take this into account, server 1 consults a special index called the *occurrence mappings*, which informs the algorithm of the location of possible join counterparts.

In our example, since b is our current value for $?X_2$, this index tells server 1 that triples containing b in the subject position occur in both server 1 in P_1 and server 2 in P_2. Therefore, server 1 sends the partial binding μ to server 2, instructing server 2 to match the second triple pattern in P_2. Server 2 thus extends μ by mapping $?X_3$

to d, which produces one query answer. Moreover, server 1 also matches the second triple pattern in P_1; that is, it extends μ by additionally mapping $?X_3$ to c and thus produces another query answer. No further matches are possible, so the servers use a specially designed protocol to inform each other that query evaluation is complete.

While this summary omits several details, it highlights certain important properties of the approach. First, servers use a special index to locate join counterparts; thus, as long as the index is correctly populated, the algorithm can be used with any data partitioning strategy. Second, partial bindings correspond to the answers of the prefixes of the triple patterns; in our example, μ is an answer to the prefix containing just the first triple pattern. Third, distributed joins are processed by moving computation, rather than moving data: in our example, join processing 'hops' from server 1 to server 2. Fourth, processing is asynchronous: a server can process messages with partial bindings as soon as they are received, independently from other servers. As a result, this approach outperformed TriAD and AdPart on certain data and query loads [25].

3

Partitioning in Distributed Architectures

Due to the versatility of RDF, the structure of a graph can grow in any direction resulting in any shape of graph. As such, there is no one-size-fits-all partitioning that will work well with every structure of an RDF graph. Data Exchange during distributed query answering is costly and is commonly seen as the bottleneck for a distributed system. If the query answering algorithm is aware of the placement of data, it could use this information to reduce the amount of data exchange during evaluation which in turn translates to a reduction in network communication. A partitioning strategy could provide some useful guarantees that the query planner could take advantage of.

Wylot et al. [6] have recently surveyed the state of the art in both centralised and distributed RDF stores, and Abdelaziz et al. [24] have presented a detailed survey of the distributed RDF stores. Prominent distributed RDF stores include YARS2 [11], 4store [12], H-RDF-3X [13], Trinity.RDF [14], SHARD [15], SHAPE [16], AdPart [18], TriAD [19], SemStore [20], DREAM [21], WARP [22], Partout [17],

and RDFox [25], to name just a few. In this section, we briefly summarise the data partitioning methods commonly used in these systems.

To partition RDF data, most systems aim to place triples with the same subject on the same server. This is motivated by an empirical study [30] of over three million SPARQL queries, which showed that approximately 60% of joins in queries are subject–subject joins, 35% are subject–object joins, and less than 5% are object–object joins. Thus, by assigning triples with the same subject to the same partition element, no communication is needed for the most common type of join, which is key to the performance of distributed RDF systems [13]. In this chapter, we look at a few partitioning techniques for RDF data.

3.1 Hash Partitioning

This is the most used partitioning technique where a hash function is applied to a key and the result is used to determine the appropriate container. Hash partitioning is easy to use however, the particular keys and hash function chosen could result in uneven distribution of data across the partitions. Consequently, redistribution of data could be difficult and expensive. If for instance, another server is added to the cluster, the whole database has to be reorganized with the new hash function being applied to each data piece before being re-allocated. While *subject hashing* partitioning was first introduced in the YARS2 system [11], it is currently the most commonly used partitioning strategy for RDF [11, 14, 31, 32]. To partition an RDF graph G into ℓ elements, each triple $\langle s, p, o\rangle \in G$ is stored at server $(h(s) \mod \ell) + 1$, where h is a hash function mapping the triple's subject to an integer.

Subject hashing is very popular since it requires no communication for subject–subject joins, it can easily be implemented in streaming fashion, and it typically produces very balanced partitions. Whether the partitions are balanced or not is largely dependant on the hash function used. With this partitioning strategy, the triples that have the same resource in the subject position will be located on the same server and thus guarantee locality of all subject–subject joins. Due to

an empirical study [30] of over three million SPARQL queries, which showed that approximately 60% of joins in queries are subject–subject joins, subject-subject joins are believed to be the most important join to guarantee for distributed RDF stores. The main drawback is that communication is usually needed for subject–object or object–object joins.

Subject hashing can be extended further by adding either partitioning by object or predicate too. One variation of this is the double hashing done in TriAD [19] which hashes triples by subject but also hashes triples by object, thus increasing the likelihood that triples participating in a join are colocated. While this guarantees locality of both subject-subject and object-object joins, this effectively duplicates the amount of data in the system, which can be prohibitive as it increases the storage overhead.

3.2 METIS Min-Cut Graph Partitioning

The RDF partitioning methods we discuss in the rest of this chapter use the min-cut graph partitioning algorithm from the METIS library [23]. The min-cut algorithm is not directly applicable to RDF: it accepts as input a directed, unlabelled graph $G = (V, E, w)$ consisting of a finite vertex set V, an edge set $E \subseteq V \times V$, and a weighting function w that assigns to each vertex $v \in V$ a nonnegative weight $w(v)$. The algorithm aims to split the set of vertices V into mutually disjoint sets $V_1, \ldots, V_n$ such that (i) $\sum_{v \in V_i} w(v)$ is roughly the same for each V_i, and (ii) the number of edges connecting vertices in V_i and V_j is minimised. The decision version of this problem is NP-hard. However, the min-cut algorithm in METIS has been highly tuned to efficiently produce good solutions in practice.

3.3 n-hop: Vertex Partitioning with Duplication

To reduce complexity during join evaluation, [13] introduces n-hop partitioning that uses METIS for subject vertex partitioning coupled with data duplication to

increase locality of answers in partitions. In particular, this technique first removes all triples whose predicate is *rdf*:*type*: as the objects of such triples typically have many incoming edges, which can confuse METIS. The resulting graph is straightforwardly converted into a directed, unlabelled graph, where the weight of each vertex is set to one. The vertices of this graph are partitioned using METIS into ℓ partitions $V_1, \ldots, V_\ell$. Finally, each triple $\langle s, p, o \rangle \in G$ is assigned to a partition element P_k such that $s \in V_k$. Intuitively, this approach minimises the number of triples spanning partitioning elements, which should in turn minimise communication during query answering. To further reduce communication, this approach also replicates data: for a chosen integer n, the approach adds to each partition element P_k all triples of G that are necessary to ensure that any query containing n 'hops' (i.e., joins whose computation requires traversing at most n connected vertices) can be evaluated without any communication.

The introduction of duplicate triples could be effective at reducing communication, however, it can be impractical as it could incur a significant data storage overhead [13] depending on the dataset and the value of n. For example, it works extremely well in hierarchical tree-shaped graphs like that demonstrated in the LUBM dataset [27] but a highly cyclic dataset like WatDiv [28] does not fare as well. We tested this and found that while the n-hop partitioning algorithm with a hop of 2 works well with LUBM500, however the duplication overhead when run on WatDiv500 resulted in the dataset increasing to 603% its original size with most of the partitions produced almost entire replications of the original dataset. A variant of this approach has been used in the D-SPARQ system by Mutharaju et al. [33].

3.4 Weighted vertex partitioning with pruning

Weighted vertex partitioning with pruning [25] is based on METIS. Just like in *n-hop partitioning*, it also first prunes the given RDF graph by removing all triples of the form $\langle x, \textit{rdf}{:}\textit{type}, y \rangle$, as well all triples of the form $\langle x, p, \textit{lit} \rangle$ where *lit* is a literal: such triples are also prone to introducing hub vertices, but they rarely

participate in object–object joins. The resulting graph is partitioned using METIS, but the weight of each vertex v is set to the number of triples of G that have v as subject. Since the METIS algorithm aims to make the sum of weights of vertices in each partition roughly the same, this effectively ensures that final partition elements are of roughly the same size. Finally, triples of the input RDF graph are assigned to partition elements as in the previous paragraph, but without any replication.

3.5 Triple Grouping

Triple grouping introduced in SHAPE [16] is a partitioning algorithm that is similar to that of n-hop guarantee with the difference that there is no specified value for n. For each resource in the graph, a triple group is created and these groups are allocated to partitions. There are three main types of triple groups with differing query evaluation guarantees; a subject-based triple group, object-based triple group, and a subject-object-based triple group. A *subject-based triple group* is one where the anchor resource is a subject, so all the triples where this resource occurs in the subject position is put in the same triple group. For the *object-based triple group*, it follows the same principles, only this time all the triples that contain the anchor resource in the object position are added to the same group. Finally, for the third type, the *subject-object-based triple group*, we pick all the triples that have the anchor resource in either the subject or the object position and assign them to the same group.

The allocation of groups is in the manner that achieves both partition balance as well as reduces resource overlap and duplication as much as possible. The anchor is the resource used to build the triple group. The triple groups are allocated to partitions usually by running a hash function on the anchor resource. While the subject and object based triple groups could be allocated without any duplication, the subject-object-based triple groups, have the potential for duplicate triples after allocation, so the allocation strategy tries as much as possible minimise the overlap of triples between partitions. While this approach can work really well with a

hierarchical dataset such as LUBM [27], it doesn't work as well for more complex, real world datasets such as WatDiv [28].

3.6 Rooted Subgraphs

A complex query, is a query that may contain more than one join of different types. Complex queries are common in practice and evaluating them over a distributed architecture that used n-hop guarantee or triple groups might be difficult. This is because for the triple groups, we only extend one edge from the anchor resource. For the n-hop guarantee, you could extend to a specified number of n but due to the replication overhead, it is usually used with smaller values of n. An alternative to these approaches is introduced in SemStore [20] which partitions using rooted subgraphs. A rooted subgraph is similar to a subject-based triple group but instead of just getting all the triples where the resource is a subject, it gets all outgoing directed paths from the root resource. The root node is similar to the anchor resource for triple groups.

Partitioning by rooted subgraphs could be done in such a way that there is a rooted subgraph for each of the nodes in the graph and they are allocated in such a way so as to minimize data replication, another option would be to carefully select which nodes are going to be root nodes in such a way that the union of all the rooted subgraphs is the full original graph. Similar to triple groups, rooted subgraphs could work well for a structured, hierarchical graph such as the LUBM dataset [27], however, this is a synthetic dataset and real life datasets are usually more complex. Generating rooted subgraphs for a highly connected graph could result in significant replication that could in turn cause a storage overhead.

3.7 Conclusion

In this chapter we have discussed different partitioning strategies. However, the best approach for any dataset depends on a lot of factors. In Chapter 5 we will introduce two of our own partitioning algorithms and we will assess their performance against the current state of the art. We will then test the performance of these partitions against other state of the art partitioning strategies in the evaluation Chapters 7, 8 and 9.

4

Distributed Query Answering

Data exchange is one of the biggest challenges for distributed RDF systems. Since the servers in the distributed architecture do not share data stores, the data is distributed among the machines, and when a query is run, query evaluation could need data stored on several of the servers. In this chapter we explore the current solutions for data exchange in most distributed systems with interest in how they work, the challenges they face, optimisations done to tackle the challenges, where they excel and where they fall behind other approaches. We conclude the chapter by justifying our choice for the query evaluation algorithm we use to evaluate our approaches.

4.1 Data Exchange Operator

The Volcano system [34] first introduced the data exchange operator that can be used to assist with parallelism between servers hosting partitions of a dataset. It is so robust that it can be used for parallel execution of processes on a single server

or for execution on a distributed system that spans across servers. It encapsulates communication between processes and facilitates the transfer of information between processes such that each process has all the information required for successful query evaluation.

The requirement for the insertion of a data exchange operator depends on the type of partitioning used (query guarantees) and the query being evaluated. For instance, if a conjunctive query has an object-object join and the data was partitioned in such a way that all triples with the same subject occur on one partition, we cannot guarantee that all the triples required to match the atoms in the object-object join are located on the same server. In such an instance, data exchange operators are inserted in the query plan before the join evaluation. The data exchange operator will transfer the intermediate answers for the matches that have been evaluated thus far and sent to the destination server where evaluation can continue.

Most modern distributed systems have incorporated the use of variants of the data exchange operator in their query answering algorithms. The main reason for this is the need for communication between the servers in the distributed architecture. While data exchange operators help facilitate the exchange of data between servers, how and when they are used could significantly affect query evaluation performance. As a result, the query evaluation plan needs to be devised in such a way that communication is limited and the query atoms are evaluated in an order that promotes computation of local answers on the different partitions.

For a particular partition P_k, local answers are answers that can be matched locally on P_k without the need of information from other partitions, non-local answers are the ones that reside on different partitions. If a query is evaluated independently on the different partition elements, the query will produce local answers only. In order to complete the full computation of the query, data exchange operators have to be inserted between two operators so as to ensure that non-local answers are also produced.

In order to achieve this goal, query plans reorder query atoms to take advantage of the partitioning technique used based on the guarantees provided by the partitioning

algorithm, decide where data exchange operators can be inserted. In most systems, the decision of where to place data exchange operators is made statically during the formation of the query plan. This could result in unnecessary transfer of information during query evaluation for the computation of local answers. Since the answers are local, the data does not have to be moved as the answer can be computed completely on the host server which could result in significant communication and storage overhead between the servers.

This redundant transfer of information could affect scalability by increasing inter-partition communication which we are trying to avoid. We produce strong partitions that reduce the need for distributed join processing during query evaluation and as such, we need a query evaluation approach that will limit unnecessary communication between the servers.

4.2 Existing RDF Systems

In this section, we discuss some key distributed RDF systems. Some extensive surveys have been conducted on the current state of the art which are in the literature [35] [36] [37]. The systems are classified in three main categories based on their data partitioning strategy as well as their mode of storage.

The three categories are as follows:

1. Key-Value Stores: These are systems that store the data in key-value stores after performing some form of changes to the data to make it suitable for the system,

2. Centralized RDF Stores: These systems run an independent RDF store on each server, and

3. Distributed RDF File Systems: They give you a local view of the data while they abstract away the storage details.

For each of these systems, we will also look at their query evaluation protocols. We conclude this section by discussing in more detail the approach we chose to use for evaluation as we require a uniform environment to test the partitioning strategies developed in our work.

4.2.1 Systems Using Key-Value Stores

Key-value stores have become a popular branch of NoSQL databases that use an associative array as their data storage model. There are variations of this kind of data store with some approaches being more complex than others. A decision is made on what values will be used as the key, most of the time being a subject and could be a combination of other characteristics. The choice on the data structure for the storage of the values is usually one that is custom built for the system. Since the data model used to store the data is custom built, the systems have their own query evaluation protocols, algorithms and operations.

Trinity.RDF

Trinity.RDF [14] is an in-memory distributed RDF store that firstly partitions the data using a hashing function. The RDF data is then stored as a graph in a key-value store. It is an improvement on Trinity [38].

H2RDF+

H2RDF+ [39] is an improvement of H2RDF [40] which is built on HBase [41]. It is a distributed RDF store that is also classified as a NoSQL data store. It has an extensive indexing scheme. It is a key-value store that sorts its data using six indexes based on the different possible combinations of the triple elements first introduced in Hexastore [42]. As a result, join evaluation is performed quickly. However, due to the multiple indexes of the data, this system incurs significant storage overhead as the data is duplicated. It uses MapReduce combined with merge and sort-merge joins during query evaluation.

Yars2

Yars2 [11] is interesting as it stores the data in a key-value data store, though its data is stored as quads instead of triples. The key in the store is made using various indexes over the data and the key is selected as one of the elements of the quad. To partition the data, Yars2 uses a hash function and uses index nested loop joins for the query evaluation.

EAGRE

EAGRE [43] partitions the data as a graph and generates what they call RDF entities. An RDF entity is a key resource and the set of all the resources that are reachable one hop from the key resource. The key resource is used as the key in the key-value data model with the set of all its neighbours as the values. In a nutshell, it partitions subjects and for each subject, the value is the set of objects where the key resource occurs in the subject position. Since RDF data is inherently connected and a resource could occur in the subject position for more than one triple, the object groups could be overlapping which could contribute to storage overhead.

4.2.2 Systems Using Centralized RDF Stores

These type of systems use a centralised RDF store on each of the servers in the cluster and these servers communicate during query evaluation with each other using a message passing mechanism. The data is partitioned using a partitioning strategy of choice and each server is assigned a piece of the data. Each independent server is responsible for storing, querying and indexing its assigned subset of the data.

n-hop Duplication System

Although this is one of the first implementations of such an RDF store, it works by using a combination of systems and was developed by Huang et al. [13]. They start by partitioning the dataset using METIS [23] that splits the dataset into k parts with one partition per server. METIS has a graph partitioning

algorithm that aims to produce balanced partitions whilst minimising the number of edges cut. After partitioning the subjects using METIS, some duplicate triples are added depending on the value of n. The n-hop guarantee has already been explained in detail in Chapter 3.

After partitioning, each server is given one of the partitions and the partition is stored in the centralized RDF store RDF-3X [10]. During query evaluation, the query is broken down into shorter subqueries such that each of the subqueries can be evaluated locally without the need to get some answers from other servers. All the local answers produced by the different servers are then joined using Hadoop [44].

TriAD

TriAD [19] stores its data using an index-based triple store. It partitions the data by double hashing, that is, it hashes by both the subject and the object. This essentially doubles the graph size and may pose a storage overhead. To avoid unnecessary matching of triples, they use graph summary technique to implement join-ahead pruning to eliminate partitions that contain no matching triples to satisfy the query. This reduces unnecessary server workload and could potentially reduce query evaluation times as well. During query evaluation, to facilitate asynchronous communication, TriAD uses their own implementation of a Message Passing Interface (MPI).

Splendid

Splendid [45] is an interesting take on an RDF system as it encourages integration of already existing systems. It is a federated approach where there are no restrictions on the RDF database to use. The only requirement is that each of the RDF databases in the cluster should have a SPARQL endpoint. Splendid is more of integration of different systems as opposed to partitioning and a such, it doesn't have a partitioning strategy.

PEDA

PEDA [46] is also a combination system. They partition the data using a hash partitioning strategy on the triples. On each of the servers in the cluster, it uses a centralized graph-based RDF database gStore [47] to store the data. For query evaluation, the query answering problem is converted into a subgraph matching problem. The query is broken into subqueries and the servers communicate through an MPI as well.

DREAM

DREAM [21] is an interesting one as it does not partition the data at all as they use the Quadrant-IV approach which means that each server in the cluster will have the whole graph stored on disk. This effectively means no data is sent over the network besides simple coordination messages. For communication during query evaluation, the coordination messages are sent over an MPI. The data on each server is stored in an RDF-3X [10] database. Following that each server has a full copy of the graph, this approach requires that all servers must have enough capacity to store the entire dataset which may be a bottleneck especially seeing that a lot of large-scale datasets are encountered often.

SHAPE

SHAPE [16] just like DREAM also stores triples on the server using an RDF-3X database. Unlike DREAM, it actually partitions its data. They have their own partitioning strategy which they call semantic hash partitioning that is a combination of triple groups and n-hop guarantees. As is with any n-hop guarantee where the value of $n > 1$, this could lead to high storage overhead. To address this, they have several optimizations aimed at reducing data duplication across their servers.

SemStore

SemStore [20] has a partitioning strategy that partitions the data with the base unit being a rooted subgraph as previously described in Chapter 3. Rooted subgraphs, depending on the type, could have overlapping triples. These are triples that occur in more than one subgraph and they could pose as a challenge in terms of storage overhead. To reduce the levels of duplication, a k-means partitioning algorithm is used to allocate the rooted subgraphs so that subgraphs with more triples in common are stored on the same server. Semstore uses TripleBit [48] for the RDF store, with one instance running on each server in the cluster.

4.2.3 Systems Using Distributed RDF File Systems

Distributed RDF Systems take advantage of distributed file systems and abstract away the complexities of the distributed system by providing the users with a local view of the data. This includes managing the communication over the network, abstracting the data distribution so much that the user does not need to know from which server the data they are interested in resides on. The main focus for such systems is versatility, scalability and ease of use.

S2RDF

S2RDF [49] is an RDF distributed file system that is based on Spark [50] that is an in-memory cluster computing framework that runs queries using SQL. The queries are first vertically partitioned and the resulting partitions are stored in a Hadoop File System (HDFS). For S2RDF to work, the SPARQL queries are translated into SQL. To do this translation, they make use of their vertical partitioning scheme to map the SPARQL queries to their SQL equivalent which is then run on the server. Their vertical partitioning scheme is similar to that used in [13].

HadoopRDF

HadoopRDF [44] is built on Hadoop and the data that it stores on each server contains is stored in a Hadoop File System. The RDF data is partitioned with the triple as the partitioning unit. The triples are partitioned based on the type and predicate. When queries are run, each server produces its own local answers, and the query joins are performed using MapReduce.

Sempala

Sempala [51] is a distributed file system that is also built on Hadoop and thus stores its data in tables provided by the Hadoop File System. Sempala is similar to S2RDF in that it uses an SQL query engine Impala and hence all the queries have to be translated from SPARQL to SQL so they can be executed by the Impala querying engine.

RAPID+

RAPID+ [52, 53] is also a distributed file system for RDF that used Hadoop. During query evaluation, the local answers are combined using a MapReduce job. However, they aim to reduce the number of MapReduce jobs needed to evaluate joins. They do this by using techniques that increase parallelism between the servers in the cluster.

SHARD

SHARD [15] is also built on Hadoop like most of the systems in this category. The data is stored in a Hadoop File System. For query evaluation, SHARD uses what they call a clause-iteration approach where a query is broken down into clauses and each of these clauses is assigned to its own MapReduce job.

PigSPARQL

PigSPARQL [54] may be a distributed file system built for RDF that is built on Hadoop and stores its data in triple-based files. It however differs from the other

approaches in how it processes the queries. During query evaluation, the queries are translated from SPARQL to Pig Latin which is a query language supported by Hadoop. During run time, the triple-based files are also translated into the Pig Latin data model to allow for query evaluation.

RDFox

RDFox [25] is an in-memory RDF triple store that works as both a centralized system and a distributed system. It is flexible in that unlike most systems we have discussed in this section, it doesn't tie its query answering capabilities to the partitioning strategy. It comes with graph and hash partitioning algorithms but will work with data partitioned by another external approach.

In an effort to improve query evaluation, RDFox has a query evaluation algorithm which is highly optimised. The queries are processed atom by atom. For each atom, to avoid broadcasting the atom to all servers unnecessarily, it checks the resource map to see which servers could potentially contain matches for the atom and the atom is only sent to those servers. A *resource map* contains an entry for every resource in the dataset, the list of servers where the resource occurs in the subject, predicate, or object positions. By reading this during query evaluation, the information is used to prevent unnecessary server workload by sending the query for further processing only on the servers that contain triples that could satisfy the query.

During query evaluation, data exchange operators are used. However, unlike most systems where data exchange operators are decided statically or during query planning, RDFox uses dynamic data exchange. This means that, at any point in the query evaluation, once an atom has been matched the server will check the resource map to determine which server(s) could be used to match the next atom. By doing this, it takes advantage of local resources and processes the next atom locally if it can be satisfied locally without sending the processing to another server in the cluster. This reduces on communication between servers as well as optimises local query processing.

In addition, since the query is broken into atoms which are matched in turn, after each atom match, the current partial answer is used to do a projection to see if pursuing matching would result in a dead-end. If at any point during the matching process the projection shows that continuing matching would result in a dead end, this match is pruned or terminated to save on both time and server resources.

4.2.4 Discussion

In this chapter we evaluated the different types of RDF systems, their partitioning strategies as well as their query answering techniques. Our study is mainly to find the true effects of partitioning on RDF data with special attention to query answering. We find that most of the systems either have their own partitioning strategy, the code would have to be modified to accommodate our partitions. In addition, most of the query evaluation algorithms are not optimised to take advantage of local answers or minimize communication between the servers.

We develop two variants of our own partitioning algorithm that is described next in Chapter 5. In order for us to evaluate the true cost of partitioning, we need a distributed RDF system that:

- can host the partitions we produce,
- can be easily set up to record on each server the different aspects of the query evaluation process such as query times, matches made, partial answers produced, and the different type of messages sent per server, and
- is optimised to both take advantage of local answers and cut out unnecessary communication between the servers.

RDFox is flexible in that it can accommodate partitions produced by any partitioning strategy with minimal adjustments. It is also editable and has commands that can be used to measure different aspects of the query answering process. Finally and most importantly, their query answering algorithm minimises communication by taking advantage of local answers and for non-local answers,

computation is moved only to the servers that contain potential answers instead of broadcasting to the whole cluster.

Another reason we chose RDFox is that, Abdelaziz et al. [24] surveyed 22 and evaluated 11 distributed RDF stores on a variety of data and query loads, and they showed AdPart [18] and TriAD [19] to be the best performing. However, following this survey by Abdelaziz, RDFox [25] demonstrated that due to their dynamic data exchange and other optimisations, they evaluate queries faster than TriAd and AdPart which have both been shown previously to perform better than the state of the art in query answering.[24].

4.3 Conclusion

Our goal is to minimize communication between servers during query evaluation and as such we need a system that is able to take advantage of the partitions produced as well as accept our partitions. Most RDF systems have their own partitioning strategies and their query answering techniques are tied to these implementations. RDFox has shown to be faster than the state of the art in terms of query evaluation times [25] but it's also flexible in that it can work with partitions produced by other approaches. Since we want a uniform testing environment for all the different partitioning approaches and based on our requirements we decided to use RDFox for the evaluation of our approaches. We also compare our approaches against two other partitioning methods that are both provided by RDFox; hash and weighted edge partitioning with pruning.

5

Community-Based RDF Graph Partitioning

The partitioning problem is important in several areas of data management including multi-threaded systems, distributed systems, concurrent systems and systems that run in parallel. Partitioning is the process of deciding how to distribute, data efficiently in systems that may contain several devices, files, different uses as well as different locations. A partitioning algorithm is used to decide which set of data is stored in what instance associated with a particular location. Depending on the partitioning algorithm, some duplicate data may be introduced; while this adds an additional cost of storage overhead, it could improve the overall performance of the system.

In Chapter 3 we discussed a number of partitioning techniques that are used for distributed RDF systems. The main goal of all these partitioning strategies is to reduce distributed join processing during query evaluation. To achieve this,

different techniques employ different methods. For example, subject hashing assigns all triples with a common subject to the same server so as to guarantee locality of subject-subject joins, whereas n-hop duplication uses the min-cut graph partitioning technique and then introduces a significant amount of duplicate triples so as to guarantee completeness of queries making a path of most length n. The problem with most of the partitioning strategies is that, while they try to optimise one aspect of query evaluation like guarantee a certain type of join, the other types of joins may still require a tremendous amount of distributed join processing.

In this chapter we present our novel graph partitioning algorithm for RDF that:

(i) produces partitions that significantly reduce the need for communication between servers by orders of magnitude compared to the state of the art,

(ii) guarantees completeness of subject-subject joins and reduces the likelihood of distributed joins of other types on the partitions produced, and

(iii) are balanced in size as well as load distribution during query answering.

Our approach is based on the Louvain Community Detection algorithm. We now present two variants of our approach for partitioning RDF data. To make this chapter self-contained, in Section 5.1 we first summarise the well-known community detection technique based on Louvain modularity, which we use as the basis for our work. We go on to explain or intuition behind our decision to partition by community in Section 5.2. Then, in Section 5.3 we discuss how we adapt this technique to the problem of partitioning RDF data.

5.1 Louvain Algorithm for Community Detection

Graph analysis tasks often involve the concept of a *community*, which is intuitively a set of vertices that are more interconnected among themselves than to the remaining vertices of a graph. This notion can be captured using the concept of graph *modularity*, which has been formalised by Newman [55] as follows.

Consider a graph with n vertices where a nonnegative weight of the connection between vertex i and vertex j is given by $A_{i,j}$. We assume that the graph is undirected, so $A_{i,j} = A_{j,i}$ for all $1 \leq i, j \leq n$. Note that the graph can contain self-loops—that is, $A_{i,i} \neq 0$ is allowed. The *degree* of a vertex i is the sum of the weights of the connections of i—that is, $k_i = \sum_{1 \leq j \leq n} A_{i,j}$. Finally, $m = 0.5 \times \sum_{1 \leq i \leq n} k_i$ is the sum of all edge weights, where factor 0.5 takes into account that the graph is undirected so the weight of each edge is counted twice.

Now let C be a set of *community labels*, and let us assume that each vertex i has been assigned to some community $c_i \in C$. The *modularity* of such a community assignment is the fraction of the graph edges that connect vertices in the same community minus the expected fraction had the edges been distributed at random but resulting in a graph with the same node degree distribution as the given graph. This value can be computed by

$$M = \frac{1}{2m} \sum_{1 \leq i,j \leq n} \left(A_{i,j} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j), \tag{5.1}$$

where $\delta(c_i, c_j)$ is the Kronecker delta symbol—that is, $\delta(c_i, c_j) = 1$ if $c_i = c_j$, and $\delta(c_i, c_j) = 0$ otherwise. For unweighted graphs (i.e., if each $A_{i,j}$ is either 0 or 1), modularity is a number between -0.5 and 1. The modularity is positive is if the number of edges within community exceeds the number expected on the basis of chance. Given two community assignments, the one with the higher modularity better captures the community structure inside the graph. Thus, the problem of community detection can be reduced to the problem of finding a community assignment with maximum modularity.

The *Louvain partitioning algorithm* [26] is a greedy optimization method that tries to optimize the modularity of a partition of the graph. This optimisation method is comprised of two steps; the modularity optimization phase and the grouping phase. These steps are repeated iteratively until no further iterations could improve the modularity and a set of communities is produced. The exact computational complexity of the Louvain partitioning algorithm is not known but

the method has been observed to run in time $\mathcal{O}(n \log n)$ with the first iteration taking the most computational effort. Exact modularity however is know to be NP-complete [56].

Each iteration of the partitioning contains two steps. In the first step, the algorithm tries to optimize modularity locally by creating small communities. This is done by moving a vertex to a community of its neighbour where the change in modularity is highest. The increase in modularity of moving vertex i into community c can be efficiently calculated by

$$\Delta M(i,c) = \left[\frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right],$$

where $\sum_{in}$ is the sum the weights of all the edges within community c, $\sum_{tot}$ is the sum of all the weights of the edges incident to some vertex in community c, and $k_{i,in}$ is the sum of the weights of the edges from i to all vertices in community c. Thus, in the first phase, the Louvain algorithm considers each vertex i and each neighbour j of i currently in community c_j, and it calculates the increase in modularity $\Delta M(i, c_j)$. Once all neighbours of i have been considered, if $\Delta M(i, c_j)$ for some neighbour j is maximal and positive, vertex i is moved into community c_j. This process is repeated as long as possible.

In the second step, the Louvain algorithm aggregates all vertices belonging to the same community and creates a new graph whose vertices are the communities detected in the first step. The weight of an edge connecting two communities c and c' is defined as the sum of the weights of all edges between vertices i and j that were assigned to c and c', respectively, in the first phase. The algorithm then applies the first phase to this new graph, and the entire process is repeated as long as applying the first step leads to a further increase in modularity.

The notion of modularity has been extended to directed graphs [57]. One might intuitively expect directed modularity to be more suitable to RDF since RDF graphs are directed. However, note that triple patterns in queries can traverse

an edge of an RDF graph in either direction, so we intuitively expect undirected modularity to be more suitable to our purpose.

5.2 Rationale for Partitioning by Communities

In the literature, the most common unit of partitioning is by subject. Different methods can be used to determine where the subjects are allocated and all the triples that share a common subject are placed on the same partition element. We however chose to partition not by subject but by community. The rationale behind choosing a community as a unit of allocation as opposed to just a subject which is frequently the case in the literature was that we had query answering in mind.

Our intuition is that, if we assign strong communities whole to servers, we could minimize the amount of communication required to satisfy join computation during query evaluation. A *strong community* is a community where the edges are denser within a community than between communities. A query, simply put, matches a subgraph or set of subgraphs from the main dataset. Since it is a subgraph, all the components within it are connected. If a query is a conjunctive query, it contains two or more triple patterns that share at last one resource in common. Meaning, for this query to be satisfied, within the graph G there is a path that can be followed to get from one resource to the other resources required in the query. As such, we thought, by assigning whole strong communities to partitions, we are increasing the likelihood that when a query is run, the triples required to satisfy the query will be located on the same server and hence reduce the need for distributed join processing and consequently communication overhead. For community detection, we picked the Louvain Community Detection algorithm because it is lightweight, scalable, and quick. It also is easily modified to suit our requirements. In the next section we describe the two versions of our partitioning algorithm for RDF.

5.3 Our Partitioning Algorithms for RDF

In this section, we describe two variants of a new partitioning algorithm for RDF datasets that are both based on the Louvain community detection algorithm. The pseudo-code of both approaches is shown in Algorithm 1. Function Community-RDF-Partitioning accepts as input an RDF graph G and the desired number of partitions ℓ, and it produces partition elements $P_1, \ldots, P_\ell$ that can be used for distributed query answering. From a high-level point of view, partition elements are produced by first partitioning the resources of the input graph into communities using the Louvain algorithm, allocating each community to a server, and finally allocating each triple to the server containing the community of the subject. We next discuss these steps in more detail.

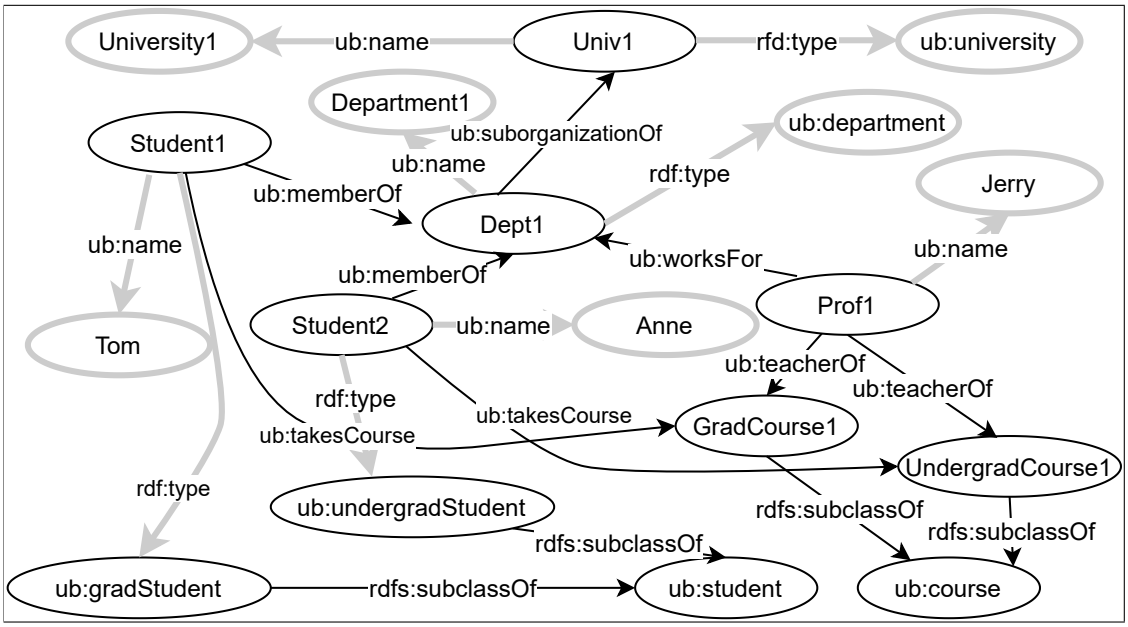


Figure 5.1: Grey edges to be pruned from sample graph

5.3.1 Pruning Phase

First, in line 2, the algorithm creates a pruned RDF graph G' by removing all triples of the form $\langle x, rdf{:}type, y\rangle$ or $\langle x, p, lit\rangle$ where *lit* is a literal. Such triples can make y and *lit* hubs, which can confuse the community detection algorithm.

Algorithm 1 Community-Based RDF Partitioning

procedure COMMUNITY-RDF-PARTITIONING(G, ℓ)
  $G' := \{\langle s, p, o\rangle \in G \mid p \neq$ *rdf:type* and o is not a literal$\}$
  Determine *maxSize* depending on the allocation variant
  $\mathcal{S} :=$ LOUVAINCOMMUNITYDETECTION(G', *maxSize*)
  for each $1 \leq k \leq \ell$ **do** $A_k := \emptyset$
  ALLOCATECOMMUNITIES
  for each $1 \leq k \leq \ell$ **do**
    $P_k := \{\langle s, p, o\rangle \in G \mid s \in A_k\}$

procedure ALLOCATECOMMUNITIES-TIGHT
  for each $T \in \mathcal{S}$ **do**
    $\mathcal{O}_T := \{o \mid \exists s, p : \langle s, p, o\rangle \in G'$ and $s \in T\}$
  for each $1 \leq k \leq \ell$ **do** $R_k := \emptyset$
  while $\mathcal{S} \neq \emptyset$ **do**
    for each $T \in \mathcal{S}$ and **each** $1 \leq k \leq \ell$ **do**
      if $|R_k \cup T \cup \mathcal{O}_T| \leq$ *maxSize* **then**
        $rank_T^k := |R_k \cap (T \cup \mathcal{O}_T)|$
      else
        $rank_T^k := 0$
    Choose $T \in \mathcal{S}$ and k such that $rank_T^k$ is largest
    $A_k := A_k \cup T$
    $R_k := R_k \cup T \cup \mathcal{O}_T$
    Remove T from $\mathcal{S}$

procedure ALLOCATECOMMUNITIES-LOOSE
  for $S \in \mathcal{S}$ **do**
    Choose k such that $|A_k| \leq |A_{k'}|$ for each $1 \leq k' \leq \ell$
    $A_k := A_k \cup S$

The same step is used in weighted vertex partitioning with pruning [25], and in *n-hop vertex partitioning* [13]. Our sample graph in Figure 1.2 will be used as our running example. We can see in Figure 5.1 the edges that are going to be removed in the cleaning/pruning phase. Once these edges have been removed, we have G' represented in Figure 5.2 which is the graph that is then partitioned. By removing triples of the form $\langle x, rdf{:}type, y\rangle$ or $\langle x, p, lit\rangle$ where *lit* is a literal, we are able to make the graph less complicated without losing the important edges. All the triples removed in this phase are later added to the final partitions so that the final partitions $P_1, \ldots, P_\ell$ when combined will give us the full graph G. That is $\sum_{1 \leq i \leq \ell} P_i = G$. For the running example, in any of the pictures, all resources/nodes

in grey occur in more than one community. These are called *overlapping resources* as they may occur in triples in the object position in one or more servers and may also occur in the subject position in another server.

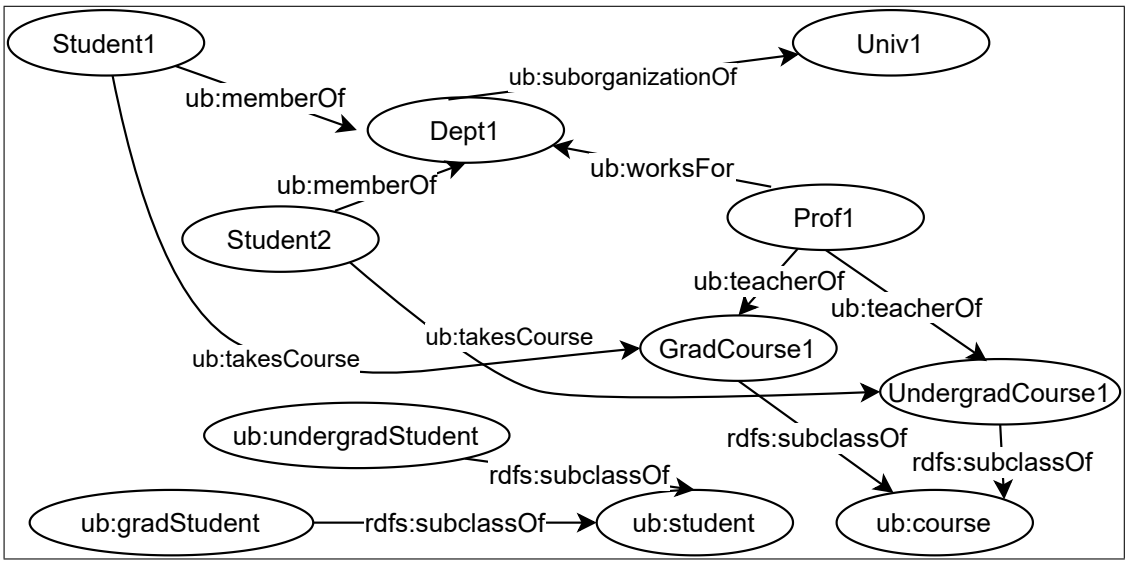


Figure 5.2: G' - Pruned sample graph

5.3.2 Community Detection

The Louvain community detection algorithm is an iterative algorithm that starts with each resource as its own community and keeps adding resources to communities as long as the move will result in an increase in modularity. In line 4, the algorithm applies a slightly modified Louvain community detection algorithm to G'. It is modified because our initial experiments revealed that the original Louvain algorithm, left to run as specified, has the potential to produce really large communities. In some of these cases, the communities that are built could even be bigger than the maximum allowed size of a partition element.

For example, if we left Louvain to run on G' as described in Section 5.1, the algorithm can create very large communities. In our running example, it would generate three communities with one of them being large and the other two containing only three resources as can be seen in Figure 5.3. In addition to that, the number of vertices in each community can vary significantly. Assigning

strong communities to servers would thus likely produce highly imbalanced partition elements, which is undesirable.

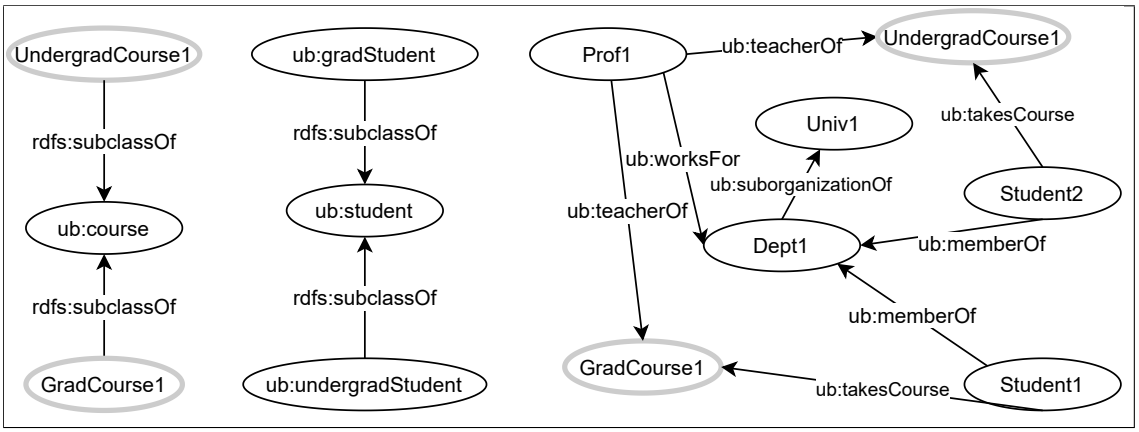


Figure 5.3: Strong communities extracted from G'

To further back our choice to limit the size of communities produced, another side effect of strong community extraction is that the most interesting part of the graph is all in one or a couple of large communities and thus if queries are run, there is a high chance that the query answers will come from the largest community that will be hosted on one server, leaving the others idle. To address this, producing strong but reasonably sized communities would work better in distributing the query workload between the servers.

In our initial experiments, the communities produced were usually so large, larger than the maximum size of a partition, so the large community would have to be re-partitioned by Louvain to break it again into smaller communities that could be easily distributed. This process is not only repetitive work but time consuming and resource wasting as the communities are built from one resource per community and in each iteration the communities are combined to make larger communities as has been described previously.

The iterative use of the Louvain algorithm was both time consuming and resource heavy. In order to produce an evenly balanced set of partitions in terms of size and workload balance, we limit the number of resources that can be added to a community. This way, each community produced is smaller than the maximum size we have set for each partition and we thus avoid re-partitioning any large

communities. Therefore, we adapted the Louvain algorithm as follows: vertex i is moved to a community c_j of a neighbour j of i only if community c_j contains at most *maxSize* vertices after the move. *maxSize* could be set to anything from the maximum number of resources the user is willing to have in a server to the maximum resources the user wants per community detected.

Our variant creates communities of at most *maxSize* elements. The value of *maxSize* is selected in line 3 depending on which community allocation variant is used in line 6, and we discuss the details in Section 5.3.3. The community detection returns a set of communities $\mathcal{S}$: each set $T \in \mathcal{S}$ contains all resources of exactly one community. Note that, $T \cap T' = \emptyset$ holds for all $T, T' \in \mathcal{S}$ with $T \neq T'$—that is, each community contains distinct resources.

In our running example graph, we run Louvain and give a $maxSize = 4$ so that we have a good distribution of a nodes amongst the communities. The final communities can be seen in Figure 5.4 where the gray nodes are the overlapping resources that occur in more than one community in either the subject or object position.

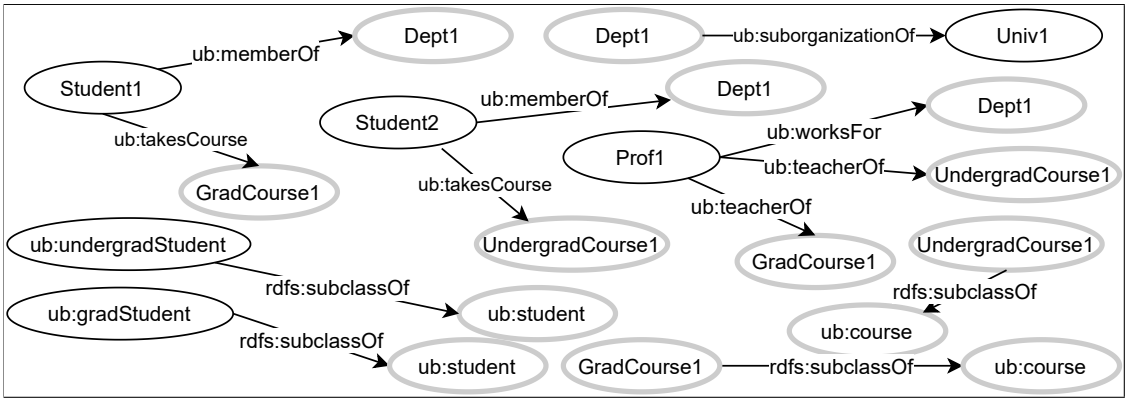


Figure 5.4: Small communities extracted from G'

5.3.3 Community Allocation

In our third stage, the algorithm assigns the resources in each community to a server. To this end, for each server $1 \leq k \leq \ell$, the algorithm introduces a set A_k that will contain the resources assigned to server k. All sets A_k are initially empty in line 5, and they are populated inside the allocation step in line 6. We developed two

variants of community allocation, tight and loose, the details of which we discuss shortly. Either way, after the community allocation step, each resource r occurring in G' in the subject position can be found in exactly one set A_k.

We next discuss our two variants of community allocation. Both take as input the set $\mathcal{S}$ of communities produced by the Louvain algorithm.

Tight Community Allocation

Although each community $T \in \mathcal{S}$ contains distinct resources, allocating T to a server will bring into the server additional resources that occur in the triples whose subjects are in T. The tight variant lets the Louvain method identify large, strongly connected communities, and it allocates communities to servers while aiming to minimise the final resource overlap. To this end, in line 3 we set $maxSize = N/\ell$, where N is the number of resources of G'.

To minimise resource overlap, for each community $T \in \mathcal{S}$, we first construct in line 11 the set $\mathcal{O}_T$ of resources occurring as objects of triples whose subjects are in T. Moreover, in line 12, for each server k, we introduce a set R_k that is analogous to A_k, but that will accumulate resources occurring in both subject and object position on server k. Then, we greedily assign communities to servers using the loop in lines 13–22.

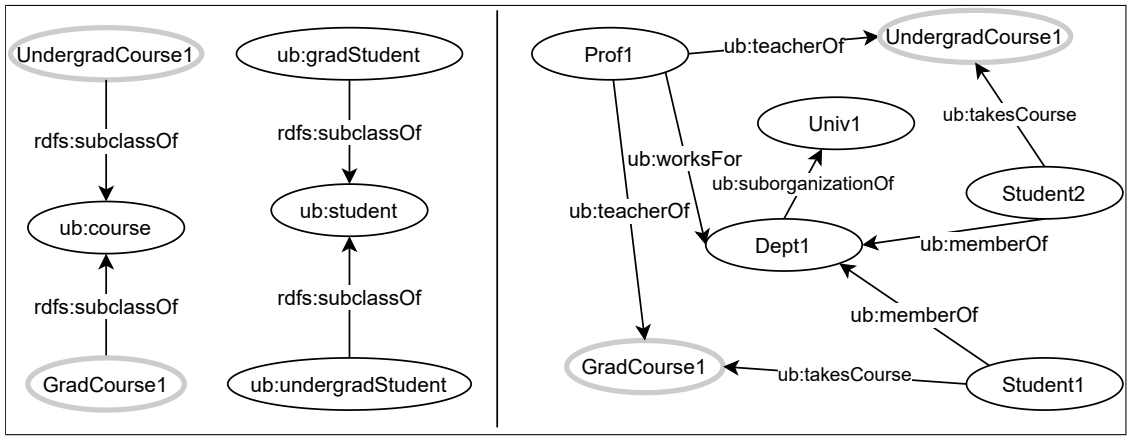


Figure 5.5: Tight allocation of communities from G'

In each iteration, the partitioner considers each unassigned community $T \in \mathcal{S}$ and each possible target server k. For each T and k, in line 16 we calculate $rank_T^k$

as the overlap between the resources R_k that are currently assigned to server k and the resources occurring in triples whose subjects are in T. In line 19, we choose the community T and the server k with the best rank (with ties broken arbitrarily), and we allocate T to k by adding T to A_k (line 20) and recording the relevant resources in R_k (line 21).

There is one final detail: if adding T to server k would assign more than *maxSize* resources to server k, then $rank_T^k$ is set to zero (line 18). This is key to ensuring that final partition elements are of roughly the same sizes (i.e., that no partition element is much larger than any other element).

For our running example, Figure 5.5 shows the tight allocation of small communities in Figure 5.4 to two servers. It starts with the community whose root node is Prof1 as it is the largest community, and allocates it to Server 1. Following the allocation of the largest partition, each following iteration only one community is picked for allocation based on the ranking. The highest ranking community is the one that has the most overlapping resources with the current allocations in the servers. Of the remaining unassigned communities, the community containing Student1 and the one containing Student2 both have two overlapping resources with Prof1's community which is our current Server 1. So we randomly pick one community since they share equal rankings and we see Student1's community added to Server 1.

In the next iteration, while Student2 community contains two overlapping resources with Server 1, adding it also to Server 1 would result in a total of 6 nodes since some of the nodes were overlapping and our total per server has been set at 7 nodes to allow for server load balance. Following this assignment, there are two communities that each have overlapping resources with Server 1. While the community containing UndergradCourse1 has two overlapping resources, the community containing Dept1 has one overlapping resource and would result in adding only one edge to Server 1 which is within the maximum limit. Therefore, the community containing Dept1 is also added to Server 1. At this point, we have a total of 7 nodes in Server 1.

Any subsequent rankings towards Server 1 will be zero as adding any more community to the server would go beyond the maximum server size of 7. The two remaining communities are added to Server 2. The result of this tight allocation is Server 1 containing a large chunk of the interesting graph, whilst Server 2 has the two island communities. The overlapping resources between the servers is only two nodes/resources (UndergradCourse1 and GradCourse1).

The final allocation is shown in Figure 5.5 with two servers, Server 1 with 7 nodes/resources and Server 2 with 6 nodes/resources. However, as can be seen from the example, with tight allocation, we tend to have really strong hubs formed in the servers and while this could reduce distributed join processing which in turn reduces communication, this could lead to a load imbalance where all the answers for a query are being generated on a small subset of the servers whilst the others are idle which could result in slower query evaluation times. This phenomenon is discussed in Chapter 7 and Chapter 8. To reduce communication whilst also distributing the workload, we developed a second variation of the algorithm which is described in the next section.

Loose Community Allocation

As we have already mentioned in Chapter 1 and as we discuss in detail in Chapters 7, 8 and 9, when an RDF graph is partitioned into highly connected communities, computational load during query answering can be focused to only some servers, which can have a negative effect on query answering times. This causes a workload imbalance as some servers are doing most of the work whilst the other are idle. As a possible solution to this problem, we developed the loose community allocation approach, which aims to strike a balance between reducing communication and spreading the computational workload more evenly to servers.

For the loose community allocation to be effective, the Louvain method should detect small communities. Since the Louvain algorithm is iterative and starts with each resource/node belonging to its own community, to ensure that the algorithm only detects small communities, we limit the community size. In line 3

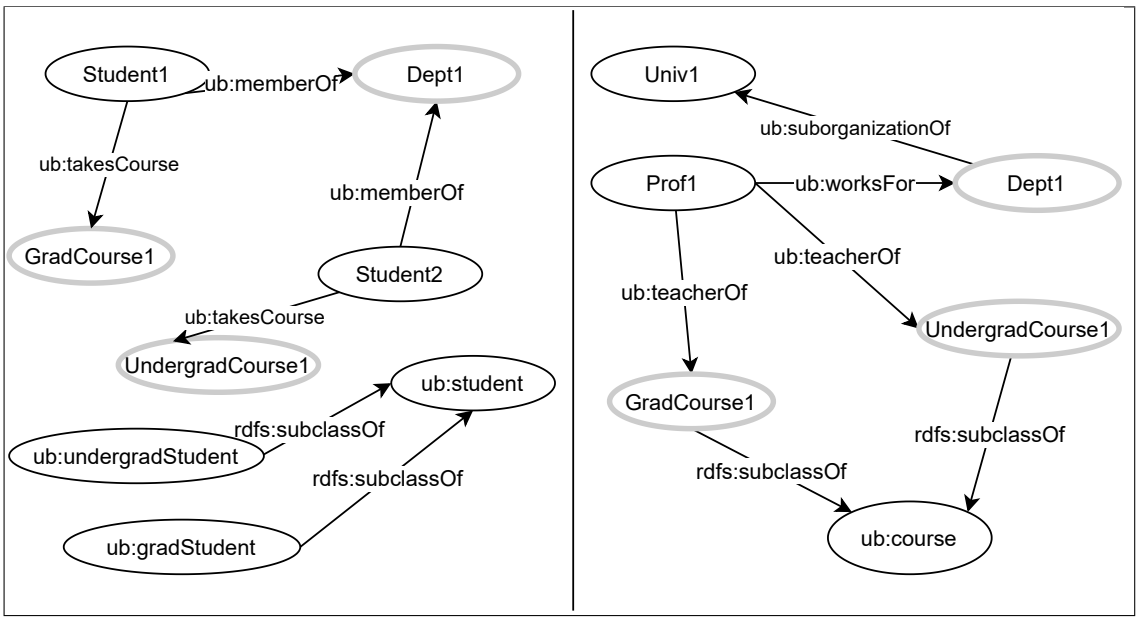


Figure 5.6: Loose allocation of communities from G'

we set $maxSize = 30$. The community size of 30 as a limit is motivated by the central limit theorem (CLT) [58] that states that the distribution of sample means approximates a normal distribution as the sample size gets larger. Sample sizes equal to or greater than 30 are considered sufficient for the CLT to hold. We therefore believe with a maximum community size of 30, we should be able to capture a sufficient subgraph that can aid in reducing distributed joins, however not too large so as to cause load imbalance.

The procedure used for allocation of these small communities for the loose community allocation is quite simple: each community $T \in \mathcal{S}$ is assigned to server k that currently has fewest resources assigned to it; ties between servers are broken arbitrarily. This is done so as to produced both balanced partitions as well as distribute the workload during query answering amongst the servers more evenly. In the evaluation it can be seen that the loose community allocation algorithm works better for highly connected datasets like WatDiv and comparably for more synthetic datasets like LUBM.

For our running example, if the communities from Figure 5.4 were allocated using the loose community allocation approach to 2 servers, the final result would be that shown in Figure 5.6. We start with allocating the largest community to

Server 1. This is Prof1's community which has 4 nodes. Following that, we have Student1's community that has 3 nodes, we assign it to Server 2 as it is currently the server with the least nodes. In the next iteration, Student2's community needs to be allocated, since Server 1 has 4 nodes and server 2 has 3 nodes, we allocate Student2's community to the server that currently has the least nodes which is Server 2 with the total nodes in Server 2 being 5.

The next community to allocate is Univ1's community which has 2 nodes. The server that currently has the lowest number of nodes is Server 1 and that is were Univ1's community is allocated bringing the total number of nodes in Server 1 to 5. The remaining communities each with two nodes each are allocated to the current server with the least nodes until all allocations are done. The final allocation is two servers, Server1 with 6 nodes and Server 2 with 8 nodes.

For both tight and loose allocation, the overlapping resources, which are the resources that occur in more than one server are shown with gray boundaries. You will notice however, that compared to the tight allocation shown in Figure 5.5 which only has only two overlapping resources (GradCourse1 and UndergradCourse1), we have three overlapping resources in the loose allocation showed in Figure 5.6 (GradCourse1, UndergradCourse1, and Dept1). This is due to the fact that we are not allocating to optimize a reduced number of overlapping resources but rather to improve on server workload distribution. We can see however, that the strong resources are fairly distributed between the servers unlike in the tight allocation where the interesting nodes are all in Server 1 and Server 2 contains mostly disjoint communities.

5.3.4 Partition Building

In our final stage we build the final partitions, the algorithm allocates in line 8 each triple of the original graph G to the partition element P_k such that A_k contains the subject of the triple. We build the final partitions from the original graph as we would like to add all triples in the partitions including those removed in the pruning phase

so that $\sum_{k=1}^{n} P_k = G$. By the end of the community detection and allocation phases, every resource that occurs as a subject in G' is allocated to a particular partition.

To minimize on memory usage, the partitioner iterates over the original graph file, triple by triple. For each triple in G of the form $\langle s, p, o \rangle$, the triple is placed in the partition k where its subject has been allocated such that $s \in A_k$. In the event that the subject occurs only in triples of the form $\langle x, rdf{:}type, y \rangle$ or $\langle x, p, lit \rangle$ where *lit* is a literal, since these triples are removed in the cleaning phase to avoid hubs that could confuse the partitioner. For such a subject s, during partition building, if we cannot find the partition k such that $s \in A_k$, then we pick a random partition j and update our allocation map so that any subsequent triples that have the same subject s will be placed in the same partition A_j.

In our running example, during the pruning phase, a total of 9 triples were removed. Fortunately, all the subjects are allocated to a partition so we don't have any island triples that need random assignment. In the tight community allocation shown in Figure 5.5 all 9 triples removed in the pruning phase are added to Server 1. In the loose allocation, 5 triples are added to Server 1 and 6 triples are added to Server 2.

5.3.5 Handling Incremental Dataset Updates

With the growing popularity of RDF, datasets will continue to grow in size. As such, datasets which have already been partitioned will occasionally have updates of new triples to be added. In both our system, we handle incremental updates in streaming fashion with no need to re-partition the entire dataset. However, how our incremental updates are handled varies slightly for the two approaches as described below.

Tight Community Allocation Dataset Updates

For the datasets partitioned by communities and allocated using the tight allocation, the goal is to minimize resource overlap as much as possible. Hence, for each triple in the update, we first check to see if the subject of that triple is already assigned to a server then this triple will be put on the same server. If not, we search for the

partition where the object has been assigned as a subject and assign it there. If neither the subject nor the object have been assigned, we allocate it to a random partition. After allocation, the resource map is updated so that future updates and query evaluation can take note of these allocations.

Loose Community Allocation Dataset Updates

The loose allocation method aims at distributing the server workload as much as possible. As such, for each triple in the update, we check the current partitioning resource map to see if the subject has already been assigned to partition and if it is, the triple is placed there. If not, we assign the triple to the server that currently has the least number of triples as we are trying to maintain partition size balance. After each triple is allocated, the resource map is updated to allow future updates and query evaluation to maximise on these allocation decisions.

5.4 Conclusion

We can thus see that for the tighter allocation, we have more pruned triples being added to the stronger partitions because they contain large chunks of the more interesting part of the graph. However, for the loose allocation, we see that there is an almost even number of pruned triples added per server as the interesting components of the graph are more evenly distributed. At the end of the partition building, all the partitions will be balanced as they will be roughly the same size and for each of the triple from G, it will be located in exactly one partition.

In the next chapters we evaluate how these two variations of our partitioning algorithm compare against the state of the art. We measure different properties of query evaluation to better understand the true effects of partitioning on query evaluation in distributed RDF systems.

6

Experimental Setting

We outline in this chapter the settings for our experimental evaluation in Chapters 7, 8, and 9. In Chapter 9 however, there is a slight difference in the hardware settings but the rest remain as is detailed in this chapter.

6.1 Test Datasets and Queries.

We evaluate our approaches on two well-known Semantic Web datasets; the WatDiv [28] and the Lehigh Universtity Benchmark (LUBM) [27]. Both of these are synthetic benchmarks however they differ in the overall graph characteristics.

6.1.1 WatDiv

The WatDiv Benchmark [28] is a synthetic benchmark that aims to simulate real-world data and query workloads. WatDiv provides a data generator, which we used

to produce an RDF graph of 1.099 billion triples. The WatDiv dataset is a high density dataset as it contains several edges between resources in its graph. Because of this, the graph is highly connected and is considered complex to partition for non-hash-based partitioning strategies that are trying to minimise inter-partition communication. While it is synthetic, it is considered in the literature to be more realistic to real-world data. As such it is often used to test the resilience and performance of systems for real-world data and workloads. It has a generator that can be used to build graphs of varying sizes which allows for testing of systems with varying specifications.

WatDiv provides 20 query templates, which have been divided into four classes: star (S), linear (L), snowflake (F), and complex (C) queries. Since star queries contain only subject–subject joins, they do not incur any communication on either of the data partitioning approaches we consider and are thus not interesting in our work. Consequently, we considered only the 13 query templates consisting of linear, snowflake, and complex queries. These templates contain a parameter that is usually replaced with a resource from the graph; however, queries obtained in this way tend to be localised and thus do not require much distributed processing. To obtain more complex and interesting queries, we replaced each parameter in a query template with a fresh variable.

6.1.2 LUBM

The Lehigh University Benchmark (LUBM) [27] was also used for this evaluation. It is a synthetic dataset that contains information about a university, its departments, students, publications, and members of staff. The size of the dataset generated is dependent on the number of universities requested for when running the generator. We used their generator to generate LUBM10K which is a dataset with 10,000 universities which resulted in 1.382 billion triples.

In literature it is often referred to as unrealistic for a benchmark because the universities have similar structures and it is basically repeating the same structures whilst making small changes to the URIs. As a result of this structure, it is relatively

easy to partition as whole universities can be placed for the most part unbroken in partitions. It is also easy to use to test systems as the size of the dataset can be easily controlled to fit the specifications of the system and thus can be used to test small-scale and well as large-scale systems. Also adding to the benefits is that it is widely used to evaluate systems in the Semantic Web community making it easy to compare new solutions against older ones previously evaluated.

The LUBM benchmark comes with 14 queries; however, without taking the ontology into account, most of them do not return any results. Thus, we used the seven queries (T1–T7) by Zeng et al. [14] that compensate for the lack of reasoning, and three new, complex cyclic queries (N1–N3) by Potter et al. [25]. All queries used in the evaluation are available in Appendix A.

6.2 Test Systems.

We used the C++ prototype by Potter et al. [25] to evaluate all queries, and to partition the data using hash partitioning (HSH) described in Section 3.1 weighted vertex partitioning with pruning (PRN) described in Section 3.4 as they are both available in RDFox[25]. We compare them against our two variations of community based partitioning; community-based RDF partitioning with tight allocation (C-T) and the community-based RDF partitioning with lose allocation (C-L) as described in Chapter 5 and we published in [59]. We reused the well-known library by Thomas Aynaud for Louvain community detection,[1] which we modified as discussed in Chapter 5 to limit the maximum community size.

6.2.1 Hardware

We performed our experiments on ten memory-optimised r5.8xlarge servers of the Amazon Elastic Cloud. Each of the ten instances was equipped with 32 CPUs and 256 GB of RAM, and it was running Ubuntu Server 18.04 LTS (HVM). The servers were connected using 10 Gigabit Ethernet.

[1] https://github.com/taynaud/python-louvain

6.2.2 Evaluation Protocol

For each of the datasets and for each of the four data partitioning techniques, we partitioned it into ten partitions. The approaches we chose to use for the partitioning all assign each triple to only one partition and as a result, all four approaches we evaluate are void of duplicate triples. Since we have have two test datasets and four partitioning techniques to evaluate, we run identical evaluations for all four resulting partition sets. For each partition in a partition set, we loaded the partition into one of the ten server such that each server hosts only one of the partitions in the set. We then proceed to run all queries while measuring wall-clock query answering times. To compensate for the variability in query answering performance, we reran each query ten times, and we report the average times.

During the evaluation we also measure the number of partial bindings messages that each server sends to the other servers per query during query evaluation, server matches made per server per query, and we report the total number of messages sent from all servers. Unlike for the query answering times, these numbers are stable across repeated runs so we measure them only once. We do not count messages needed to communicate query answers: the number of such messages should be largely the same for all data partitioning strategies assuming that query answers are distributed evenly in the cluster. In this way, the number of partial binding messages reflects the amount of communication incurred by distributed join processing, which is of main concern for our study.

6.3 Conclusion

The evaluation protocols described in this chapter are what will be used for the different evaluation Chapters 7 8 and 9. There is a slight change in the setup for our evaluation of the effects of network speed on query evaluation in Chapter 9 but any differences will be highlighted there.

7

Query Times and Communication

In this chapter, we investigate empirically how different data partitioning approaches affect the different aspects of query answering namely; query evaluation time and network communication during distributed join processing. To this end, we compared our tight (C-T) and loose (C-L) community-based partitioning variants, the weighted vertex partitioning with pruning (PRN), and subject hash partitioning (HSH); the last two techniques were described in Chapter 3. Our main objective was to see how different methods affect network communication and see the correlation with query evaluation time.

7.1 Experiment Results

The average times (in seconds) for answering all queries on all datasets are shown in Tables 7.1 and 7.2, and the corresponding total numbers of partial binding messages are shown in Tables 7.3 and 7.4.

Table 7.1: LUBM query evaluation times (s)

LUBM				
	C-L	**C-T**	**PRN**	**HSH**
T1	4.481	4.186	**3.855**	5.716
T2	4.131	**1.798**	3.441	3.903
T3	3.497	3.587	**3.445**	3.637
T4	**0.003**	**0.003**	**0.003**	**0.003**
T5	0.002	0.002	**0.001**	**0.001**
T6	**0.002**	0.003	**0.002**	**0.002**
T7	2.374	2.160	**1.905**	32.210
N1	4.341	4.049	**3.255**	26.713
N2	15.625	**12.206**	13.249	113.174
N3	2.774	**2.148**	2.194	9.869

Table 7.2: WatDiv query evaluation times (s)

WatDiv				
	C-L	**C-T**	**PRN**	**HSH**
C1	**0.157**	0.539	0.343	0.488
C2	**3.179**	9.359	8.520	5.817
C3	0.789	2.082	1.389	**0.717**
F1	1.673	5.668	3.916	**1.612**
F2	0.633	3.317	1.937	**0.477**
F3	**0.621**	3.298	1.856	0.622
F4	0.530	2.332	1.414	**0.527**
F5	**6.629**	12.315	13.029	8.858
L1	**0.733**	2.024	1.506	0.806
L2	1.189	0.847	**0.792**	1.124
L3	1.174	**1.003**	1.228	1.262
L4	0.347	1.576	0.950	**0.289**
L5	10.495	**9.095**	9,353	9.231

7.2 Discussion

As shown in Tables 7.1 and 7.2, query evaluation times vary significantly across the datasets and partitioning techniques. On the LUBM dataset, if we ignore the trivial queries T4-T6, on the seven major queries, PRN is faster on four of these queries whilst C-T is fastest for the other three. Whilst on the WatDiv dataset, C-L and HSH are fastest on five queries each, C-T is fastest on two queries, and PRN is fastest on only one query showing that the query answering performance could vary depending on the partitioning style as well as the dataset.

Table 7.3: The number of partial binding messages sent on LUBM

LUBM								
	C-L		**C-T**		**PRN**		**HSH**	
T1	2.09	k	**1.78**	**k**	72.38	k	22.67	M
T2	**10.00**		**10.00**		**10.00**		**10.00**	
T3	**10.00**		**10.00**		**10.00**		**10.00**	
T4	**0.00**		**0.00**		**0.00**		**0.00**	
T5	**0.00**		**0.00**		**0.00**		**0.00**	
T6	**0.00**		**0.00**		**0.00**		**0.00**	
T7	**10.00**		**10.00**		53.15	k	28.06	M
N1	**15.00**		18.00		184.55	k	55.85	M
N2	60.00		**58.00**		598.60	k	150.84	M
N3	**12.00**		17.00		169.26	k	37.37	M
TOT	**2.20**	**k**	**1.91**	**k**	**1.08**	**M**	**294.79**	**M**

Table 7.4: The number of partial binding messages sent on WatDiv

WatDiv								
	C-L		**C-T**		**PRN**		**HSH**	
C1	366.68	k	**350.20**	**k**	391.94	k	468.03	k
C2	4.72	M	**991.43**	**k**	4.05	M	8.97	M
C3	**10.00**		**10.00**		**10.00**		**10.00**	
F1	4.20	M	**3.09**	**M**	4.28	M	7.46	M
F2	20.53	k	**23.60**	**k**	32.88	k	33.54	k
F3	244.53	k	**128.90**	**k**	259.89	k	474.88	k
F4	88.58	k	**68.02**	**k**	98.16	k	148.10	k
F5	5.83	M	**233.53**	**k**	2.86	M	8.42	M
L1	1.94	M	**1.68**	**M**	1.86	M	2.01	M
L2	1.02	k	820.00		712.00		1.02	k
L3	**10.00**		**10.00**		**10.00**		**10.00**	
L4	**10.00**		**10.00**		**10.00**		**10.00**	
L5	54.78	M	42.89	M	**36.20**	**M**	54.93	M
TOT	**72.19**	**M**	**49.45**	**M**	**50.03**	**M**	**82.91**	**M**

We notice however, that the query evaluation times are generally close across queries and partitioning styles: the biggest difference is on query F2, where C-T is 6.95 times slower than HSH. On LUBM, if we disregard trivial queries T4 to T6, PRN is fastest on four queries, and C-T is fastest on the remaining three queries. Moreover, the query times seem to differ more than on WatDiv: on query N3, PRN is 16.9 times faster than HSH; and on queries N1 and N2, the fastest partition

approach is more than eight times faster than the slowest.

To better understand the differences in performance, we took a closer look at the amount of communication between the servers. The numbers of partial binding messages shown in Tables 7.3 and 7.4 show a completely different picture. For readability, in these tables, k represents 1,000 messages sent while M represents 1,000,000 messages sent. When an atom is matched on server k, RDFox will check to see where the next atom in the query can be matched. If the next atom can be matched locally, no messages would be sent and the next atom matched. If however, the next atom can be matched on one or more other servers, server k will send a partial bindings message to each of those servers. The message will contain information about the variables that have been matched so far from the previous atoms so that processing can continue at the other server. Because of this, we decided to measure the partial bindings messages sent as these are the means by which the servers communicate during query answering to coordinate the answers and as such a good indicator for the amount of distributed join processing. Since our interest is in measuring the amount of distributed join processing, we don't measure the answer messages sent. Answer messages could confuse our analysis as certain queries have several answers and the answer messages are sent to the coordinator node once the entire evaluation process is complete hence making them irrelevant for our analysis.

On WatDiv, C-T requires the least number of messages for all but two queries, and C-L is slightly more efficient than C-T on one more query. The difference is particularly pronounced on query F5, where C-T outperforms HSH in terms of the message number by a factor of 36. Across all queries, HSH sends 4.2 times more messages than C-T. On the LUBM dataset, the difference is even more pronounced. On queries N1–N3, C-T and C-L are about six orders of magnitude more efficient than HSH, and by around four orders or magnitude than PRN in terms of the message number.

7.3 Conclusion

These results show that community partitioning is exceptionally effective at reducing communication during join processing, but this does not always result in reduced query answering times which we found surprising considering in the literature it is believed that reducing communication between the servers during query evaluation would lead to a reduction in the query evaluation time. One interesting example is query N1 where C-L sends only 15 partial answer messages during query evaluation, while PRN sends 184.55 k messages but surprisingly PRN finishes faster in 3.255s compared to C-L that finishes in 4.341s which contradicts the commonly held conception.

We believe there are other unexpected side-effects of the partitioning processes on query answering. In Chapter 8 we look at the effects of partitioning on server workload and how this workload distribution affects the query evaluation process. In Chapter 9 we test the effects of network speed on the query evaluation process.

8

Server Workload Analysis

We have seen in Chapter 7 that the relationship between reduction in communication between the servers and reduction in query evaluation is not as straightforward as previously believed in the literature. To understand why, a closer introspection of exactly what is happening during the query evaluation process needs to be done. In this chapter, we investigate the effects of partitioning on server workload.

8.1 Experiment Results

Tables 7.3 and 7.4 show the total number of partial bindings messages sent per partitioning approach for each query evaluated however, there could be an imbalance in the number of messages sent per server. Table 8.1 shows the number of partial bindings messages sent per partition for the queries that had communication on the LUBM dataset, that is T1, T7, N1, N2 and N3. Table 8.2 shows the number of

partial bindings messages sent per partition for the queries that had communication on the WatDiv dataset, that is C1, C2, F1, F2, F3, F4, F5, L1, L2, and L5.

8.2 Discussion

In Chapter 7 we are surprised that the approaches that send the least messages by orders of magnitude compared to the others partitioning approaches are performing at par with the other approaches and in some cases is even the worst performing. We sought to find out why this is the case as it is clearly against the commonly held theory in the literature that less communication will result in faster query evaluation times. Our first step in understanding this unexpected relationship between the query evaluation time and communication is to take a deeper look at exactly how much communication is coming from each of the servers in the cluster. To do this, we analysed the number of partial binding messages sent by each server: while Table 7.3 and Table 7.4 show the messages sent for the LUBM and WatDiv partitions respectively. However, these numbers are aggregated for all servers. Tables 8.1 and 8.2 show the number of messages sent from the different servers for all the queries that involved some distributed join processing during the query evaluation. We observe that the number of messages sent per server for most of the queries tend to be fairly uniform across servers for HSH, but not for C-L, C-T, and PRN.

The behaviour of distributed join processing differs by dataset. For example, on the LUBM dataset as shown in Table 8.1, the number of partial bindings messages sent per partition is mostly uniform across the partitioning strategies. However, C-L and C-T send negligible messages, with most partitions sending no messages at all. Where fewer partial answer messages are sent, we can safely conclude that most the joins are being processed locally on the partitions without distributed processing. Conversely, PRN and HSH send orders of magnitude more messages and yet for the LUBM dataset, PRN evaluates the queries faster for four of the queries while C-T is faster for the other three main queries.

Table 8.1: LUBM partial bindings messages sent per server per query

	1	2	3	4	5	6	7	8	9	10
T1										
C-L	206	212	230	191	183	196	213	200	227	188
C-T	297	330	371	379	397	0	0	0	0	10
PRN	243	243	247	18,397	16,716	260	212	228	18,803	17,031
HSH	2,277,464	2,260,039	2,276,595	2,260,652	2,279,701	2,260,349	2,272,758	2,254,528	2,275,276	2,256,190
T7										
C-L	0	0	0	0	0	0	0	0	0	10
C-T	0	0	0	0	0	0	0	0	0	10
PRN	6,220	6,066	6,460	3,793	4,018	6,127	6,275	6,410	3,728	4,048
HSH	2,804,901	2,804,051	2,806,027	2,807,249	2,807,467	2,808,177	2,802,376	2,805,512	2,808,707	2,805,468
N1										
C-L	0	2	1	0	0	0	0	0	1	11
C-T	2	1	0	2	1	0	0	1	0	11
PRN	20,955	21,001	21,892	13,923	14,506	20,818	21,046	21,766	13,925	14,716
HSH	5,567,987	5,583,215	5,588,794	5,592,646	5,590,369	5,571,812	5,577,526	5,594,571	5,593,789	5,591,813
N2										
C-L	0	6	17	0	0	0	0	0	5	11
C-T	7	18	0	4	0	0	0	11	0	18
PRN	56,976	57,797	59,101	62,914	62,507	56,867	57,248	59,086	62,531	63,574
HSH	15,146,506	15,036,974	15,134,801	15,036,009	15,131,918	15,048,777	15,114,616	15,016,766	15,150,338	15,022,083
N3										
C-L	0	1	5	0	0	0	0	0	1	10
C-T	1	0	0	1	0	0	0	3	0	12
PRN	17,365	17,614	17,927	15,789	15,932	17,364	17,439	17,999	15,546	16,280
HSH	3,755,297	3,723,210	3,749,727	3,720,668	3,753,242	3,727,247	3,746,004	3,719,652	3,754,689	3,719,534

Table 8.2: WatDiv partial bindings messages sent per server per query

	1	2	3	4	5	6	7	8	9	10
					C1					
C-L	0	37,648	39,317	40,390	38,771	42,305	41,351	42,420	42,513	41,963
C-T	162,318	181,623	0	0	0	0	0	6,249	0	10
PRN	77,685	76,673	53,258	17,300	92,973	13,659	13,856	16,069	15,395	15,073
HSH	46,851	47,122	45,903	46,066	46,696	47,925	46,447	47,632	47,029	46,356
					C2					
C-L	0	496,088	561,110	503,128	506,940	496,387	498,093	593,184	513,434	551,667
C-T	316,776	380,903	0	0	0	0	0	293,743	0	12
PRN	533,028	514,773	749,336	270,685	1,788,238	35,493	35,700	36,240	41,746	46,506
HSH	834,238	929,021	847,813	846,620	888,314	964,939	903,655	859,817	871,589	1,020,483
					F1					
C-L	118	1,064,822	384,729	391,524	393,490	392,822	392,143	392,611	394,119	392,661
C-T	114,881	2,831,533	0	0	1	0	0	141,583	0	11
PRN	412,356	426,395	505,890	184,491	2,731,656	2,796	2,736	3,368	4,244	3,906
HSH	746,683	745,827	746,281	745,158	747,706	746,418	744,523	745,051	746,013	742,750
					F2					
C-L	6	4,485	1,923	2,006	2,054	2,034	1,999	1,985	2,023	2,015
C-T	2,963	19,723	0	2	4	4	0	773	1	12
PRN	2,981	3,105	3,960	1,108	21,171	99	88	106	111	151
HSH	3,268	3,348	3,384	3,291	3,340	3,419	3,356	3,403	3,382	3,351
					F3					
C-L	35	45,005	24,567	25,276	24,744	24,458	24,782	25,143	25,177	25,340
C-T	19,116	154,494	21	22	31	49	5	63,403	7	14
PRN	25,544	28,596	31,377	39,334	129,170	954	913	1,118	1,413	1,470
HSH	47,568	46,884	46,987	48,215	47,842	47,454	47,701	47,541	47,627	47,056
					F4					
C-L	928	11,808	9,256	9,574	9,299	9,331	9,322	9,618	9,679	9,764
C-T	1,529	56,789	363	409	2,648	4,986	47	1,104	148	133
PRN	8,740	8,738	13,087	3,168	51,470	2,023	1,901	2,315	4,450	2,263
HSH	14,723	14,969	14,588	14,823	14,695	15,720	14,718	14,730	14,646	14,492
					F5					
C-L	0	600,893	640,399	657,580	656,855	655,045	657,263	655,228	654,096	654,865
C-T	0	233,522	0	0	0	0	0	1	0	10
PRN	368,818	399,435	983,171	166	1,107,871	0	3	0	1	10
HSH	831,468	839,937	849,712	842,672	837,660	836,809	837,920	855,142	845,830	840,595
					L1					
C-L	233,276	189,747	193,729	186,814	191,268	190,423	189,895	190,827	182,355	188,789
C-T	207,572	68,717	135,429	126,445	339,119	482,591	55,726	76,310	93,375	95,809
PRN	184,887	184,923	212,950	119,303	94,896	205,515	205,527	220,105	223,602	205,704
HSH	202,853	196,230	202,185	203,119	202,582	198,869	202,050	204,627	196,364	205,136
					L2					
C-L	180	54	108	90	81	108	99	126	90	82
C-T	0	0	135	117	189	180	18	162	9	10
PRN	0	0	0	0	0	108	108	108	207	181
HSH	99	99	72	108	99	99	108	117	108	109
					L5					
C-L	9,284,853	3,145,041	5,857,086	4,954,664	4,502,592	5,857,710	5,406,268	6,763,012	4,954,953	4,051,012
C-T	346,017	61,642	7,016,169	6,131,946	9,559,368	9,110,754	1,203,924	8,345,280	756,718	353,804
PRN	214,385	214,795	201,845	228,210	55,850	5,515,968	5,509,440	5,509,880	10,234,137	8,516,787
HSH	5,403,402	5,403,112	4,052,714	5,852,421	5,403,418	5,403,688	5,853,417	6,303,080	5,853,654	5,403,390

On the WatDiv dataset, we notice a difference in the messages sent as can be seen in Table 8.2. HSH sends orders of magnitude more messages than the other approaches for nearly all the queries but we see there is an even distribution on the number of messages sent per server. This is not true for C-T, C-L and PRN, as we can see for most of the queries, these three approaches have some servers that send no messages at all during query answering whilst other servers from the same partitioning approach send many messages. One good example would be query C1 where C-T has 6 of the 10 servers sending no messages at all whilst server 1 sends 162,318 messages while HSH sends an average of about 45,000 messages sent by each of the servers. This suggested an unexpected side-effect of community partitioning: query processing tends to be confined to only some communities, which can lead to significant difference in server workload.

Table 8.3: WatDiv query L4 answers produced per server

	1	2	3	4	5	6	7	8	9	10
C-L	3	34,490	14,018	14,320	14,488	14,333	14,450	14,410	14,638	0
C-T	4,364	143,333	0	0	0	0	0	1,772	0	0
PRN	14,548	15,035	19,868	3,655	95,909	64	82	102	95	0
HSH	14,974	14,830	14,859	14,867	15,096	14,975	14,825	14,799	15,104	0

Table 8.4: WatDiv query L4 matches made per server

	1	2	3	4	5	6	7	8	9	10
C-L	37	383,566	154,880	157,785	158,217	158,292	158,446	158,347	158,726	158,196
C-T	46,791	1,579,778	0	0	1	0	0	19,921	0	1
PRN	159,234	162,861	219,005	40,874	1,059,664	805	798	1,016	1,145	1,090
HSH	164,829	164,495	164,614	164,431	165,021	164,925	164,477	164,406	165,053	164,241

To verify the above hypothesis, we measured the workload of all servers. Measuring the number of messages sent per server gives us an idea of which servers are busy but not an accurate representation of exactly how much work is done by each of the servers. This is because the messages sent are partial answer messages sent when the processing cannot be continued locally on a server anymore. Intuitively, we thought to measure the number of answers produced per server but on further observation, this too does not give an accurate representation of the amount of work done per server. This is because, to produce a single answer, several atoms have to be matched on the servers and the matching process is time

consuming and even though not all matches result in answers, they contribute towards overall server workload.

An example is query L4 from WatDiv; Table 8.3 shows the number of answers produced per server per partitioning approach, whilst Table 8.4 shows the matches made per server. We can observe that the skew on matches per server is more harshly seen. For example, 5 of the C-T servers make absolutely no matches, with 2 servers only making a single match. Essentially, the entire query is matched from only 3 of the 10 servers with Server 2 alone making 1,579,778 matches. On the other hand, for HSH, each server makes an average of 164,000 matches. This shows that for the partitioning strategies that produce very strongly connected partitions, the query processing tends to be restricted to only a few servers whilst the other servers remain mostly idle. The opposite happens for HSH, since the dataset is evenly distributed, while this results in increased distributed query processing, the servers have a distributed load. Based on this initial observation, we test to see the relationship between server workload and query evaluation further by measuring the matches made per server.

The distributed query answering approach by Potter et al. [25] evaluates queries by matching atoms using index nested loops joins; thus, the number of times an atom is matched at each server provides a good proxy for the server workload. We modified the RDFox prototype to capture the number of atom matches, and we reran all our test queries. The number of matches on each server does not depend on the exact run, so we run each query just once. Moreover, the total amount of work (i.e., the total number of atom matches) depends only on the input RDF graph, and not on the data partitioning technique.

Tables 8.5 and 8.6 summarise the results of this experiment for WatDiv and LUBM, respectively. Showing the matches for each of the ten servers would be inconvenient so, for each query and partitioning technique, we show only the maximum, minimum, and median number of matches across all servers.

These results confirm our initial hypothesis. On WatDiv, C-L and C-T incur considerable difference in server workload on most queries. This difference also

Table 8.5: Atom matches for WatDiv

	C-L						C-T					
	MAX		**MIN**		**MED**		**MAX**		**MIN**		**MED**	
C1	85.24	k	47.00		59.27	k	420.55	k	2.00		58.00	
C2	2.03	M	18.79	k	1.92	M	14.32	M	13.17	k	50.95	k
C3	789.06	k	573.18	k	577.96	k	871.84	k	190.71	k	597.10	k
F1	1.68	M	884.00		825.62	k	7.60	M	29.00		683.50	
F2	822.87	k	4.23	k	466.56	k	4.02	M	209.00		8.16	k
F3	678.19	k	3.63	k	407.05	k	3.08	M	1.66	k	7.99	k
F4	506.83	k	15.60	k	220.86	k	2.08	M	7.54	k	15.56	k
F5	3.12	M	500.00		3.05	M	16.89	M	8.00		255.00	
L1	476.34	k	433.13	k	439.02	k	930.26	k	171.98	k	339.63	k
L2	1.85	M	1.37	M	1.36	M	1.40	M	183.15	k	956.57	k
L3	476.32	k	413.02	k	416.77	k	930.23	k	171.97	k	321.37	k
L4	383.57	k	37.00		158.25	k	1.58	M	0.00		0.50	
L5	10.12	M	3.12	M	5.37	M	10.63	M	17.68	k	3.85	M
	PRN						**HSH**					
	MAX		**MIN**		**MED**		**MAX**		**MIN**		**MED**	
C1	271.20	k	5.86	k	13.30	k	57.38	k	54.74	k	55.79	k
C2	9.55	M	91.07	k	473.47	k	1.85	M	1.68	M	1.72	M
C3	829.22	k	175.83	k	632.51	k	600.23	k	597.52	k	599.47	k
F1	5.08	M	18.67	k	144.93	k	828.11	k	824.86	k	827.55	k
F2	2.66	M	23.04	k	102.81	k	456.38	k	453.59	k	455.10	k
F3	2.01	M	27.41	k	219.02	k	393.71	k	392.30	k	393.20	k
F4	1.39	M	15.75	k	44.02	k	229.54	k	227.40	k	228.90	k
F5	12.87	M	12.47	k	138.87	k	2.83	M	2.78	M	2.81	M
L1	505.46	k	391.51	k	446.57	k	449.26	k	440.29	k	443.58	k
L2	1.14	M	205.81	k	912.82	k	1.43	M	1.43	M	1.43	M
L3	504.70	k	302.83	k	439.79	k	426.24	k	420.57	k	422.85	k
L4	1.06	M	798.00		21.01	k	165.05	k	164.24	k	164.55	k
L5	11.63	M	21.28	k	3.10	M	6.62	M	4.12	M	5.62	M

Table 8.6: Atom matches for LUBM in millions (M)

	C-L			C-T			PRN			HSH		
	MAX	**MIN**	**MED**	**MAX**	**MIN**	**MED**	**MAX**	**MIN**	**MED**	**MAX**	**MIN**	**MED**
T1	7.57	7.55	7.56	7.77	7.42	7.47	7.80	7.21	7.78	7.59	7.53	7.56
T2	2.16	2.16	2.16	2.22	2.11	2.13	2.24	2.05	2.23	2.16	2.15	2.16
T3	3.24	3.24	3.24	3.33	3.18	3.20	3.34	3.09	3.34	3.25	3.23	3.24
T7	3.08	3.07	3.07	3.11	3.05	3.06	3.18	2.92	3.17	3.08	3.06	3.07
N1	5.15	5.14	5.14	5.20	5.10	5.12	5.31	4.89	5.31	5.16	5.12	5.14
N2	26.95	26.80	26.88	27.52	26.42	26.62	27.72	25.63	27.66	26.99	26.76	26.87
N3	3.54	3.53	3.53	3.63	3.46	3.50	3.65	3.36	3.63	3.55	3.52	3.53

exists with PRN, but it is somewhat less pronounced. Finally, there is almost no difference in the workload for HSH, which is unsurprising given that triples are spread uniformly across the cluster. In contrast, the workload was roughly the same for all data partitioning techniques on LUBM.

Thus, query answering times seem primarily determined by the following:

1. the distribution of the workload in the cluster, and

2. the communication cost.

Thus, if the communication cost of the two strategies is roughly the same, then

the strategy with a more even workload distribution is more efficient: the total amount of work is the same, so the query answering time depends on the time of the slowest server. This seems to be the case on WatDiv: C-T requires only about 4.2 times fewer partial binding messages than HSH, but this improvement is insufficient to offset the stark difference in server workload. As a result, HSH often outperforms PRN and C-T on WatDiv in terms of query times. This is particularly pronounced on queries F1–F5 and L4, where the gaps in both the performance and workload distribution seem significant.

It is interesting to note however the stark difference in performance between C-T our tight community allocation approach and C-L our loose community allocation approach. These approaches only differ by allocation strategy with one minimizing resource overlap therefore producing strong partitions, and the other randomly allocating communities so as to improve query workload balance.

On LUBM, C-T performs better than C-L on all but two of the queries but the opposite is true on WatDiv, where C-L is faster than C-T on all but two queries. These results are influenced by two main factors; the dataset type, and the workload distribution resulting from the partitioning strategy.

WatDiv is a very dense dataset as the number of edges per resource is high compared to LUBM that is relatively sparse. As a result, tight allocation results in extremely dense partitions and query evaluation is done by a small subset of the servers which leads to load imbalance. So the key factor determining who finishes faster on WatDiv is mostly dependent on load balance.

Conversely, when we look at LUBM, since its a sparse dataset, it is easy to partition and we see from Table 8.6 that each server makes roughly the same amount of matches which means that every partition has the workload balanced. However, now communication comes into play, C-T allocates communities to minimize resource overlap and reduce communication, as a result, we see communication working as the tie breaker and thus C-T works better for LUBM.

We have seen from the evaluation that workload distribution could drastically affect query evaluation time more so than reduction in communication. However,

if workload distribution is roughly evenly distributed, then the strategy with less communication is more efficient. This seems to be the case for LUBM: all four data partitioning strategies distribute the workload in roughly the same way; however, C-L and C-T incur significantly less communication than HSH on queries N1–N3, which seems to correlate with a significant boost in query performance. Although C-L and C-T incur significantly less communication compared to PRN, the absolute numbers of partial answer messages for PRN are quite small, and so the performance in these cases seems to mainly depend on other factors.

8.3 Conclusion

In this chapter we took a deeper look into the query evaluation process in order to understand the reason why our approach C-T that sent a total of 1.91k messages was performing at par and sometimes worse than other approaches that sent a total of 1.8M and 294.79M messages for PRN and HSH respectively. This behaviour was inconsistent with the commonly held beliefs promoted in the literature that less messages sent would result in faster query times.

We look at the total messages sent for each query but this does not give us a clear understanding of why this is the case so we compared the messages sent per server per query and noticed that for the approaches that sent less messages, most of the computation was being done on a few of the servers whereas the other approaches balanced computation between the servers. As a result, the approaches that sent the fewest messages, while their servers were balanced in size, their workload during query evaluation was highly imbalanced.

Server workload imbalance is more stark when we looked at the total matches made per server and this phenomenon is attributed to the fact that the approaches like C-T that sent less messages in general had strong graphs in the partitions where the connections between the resources in the partition were higher. As a result, most of the atoms required to satisfy a query would reside on a few of the servers leaving these selected few servers to do all the work whilst other servers in

the cluster remained idle. However, approaches like HSH that also had partitions balanced in size, also had somehow evenly distributed the triples needed to satisfy the queries evenly amongst the servers resulting in high communication but also an even workload distribution amongst the servers in the cluster. Hence, despite the multitude of messages sent, the approach still performed well and in some cases was best performing of the four approaches.

We have come to the conclusion that, whilst it's important to optimise partitioning in order to minimize communication between servers, it is more important to optimize server workload balance as this plays a bigger role in determining how fast query evaluation will take. Minimising communication between the servers is not irrelevant as we observed on the partitions produced for the LUBM dataset that in the where server workload is balanced among the different approaches, the amount of communication per approach now works somewhat as a tie breaker in the race to completion.

9

Impact of Network Speed

In this chapter we further analyse our results, we evaluated our queries in a setting where network is artificially slowed down. Please note that we do not argue that reduced network speed is realistic in practice; rather, our goal was just to see how network communication and workload imbalance affect query answering performance.

9.1 Experimental Setup

We performed our experiments by running ten instances of our prototype system on a single memory-optimised i3en.24xlarge server of the Amazon Elastic Cloud with 768 GB of RAM, running Ubuntu Server 18.04 LTS (HVM). All instances of our system communicated using the loopback interface as they were all hosted on the same server. We simulated a slow network environment using the NetEm[1]

[1]https://man7.org/linux/man-pages/man8/tc-netem.8.html

Table 9.1: LUBM query times (s) with network delay (ms)

T1 Delay	C-L	C-T	PRN	HSH		N2 Delay	C-L	C-T	PRN	HSH
0	4.110	**4.102**	4.469	5.427		0	13.250	12.734	**12.501**	82.330
1	**4.356**	4.360	4.468	8.310		1	13.471	**12.745**	12.975	255.824
10	**4.325**	4.406	4.715	25.450		10	**15.895**	16.263	18.322	291.051
100	**5.285**	5.324	5.501	211.935		100	**106.541**	109.767	121.364	3,679.016
T7 Delay	**C-L**	**C-T**	**PRN**	**HSH**		**N3 Delay**	**C-L**	**C-T**	**PRN**	**HSH**
0	**2.140**	2.141	2.147	24.141		0	2.576	**2.459**	2.474	7.727
1	2.223	**2.189**	2.201	72.757		1	**2.535**	2.645	2.585	14.814
10	**2.453**	2.534	2.581	41.346		10	4.002	**3.897**	4.314	68.820
100	9.506	**9.331**	10.116	547.175		100	**29.712**	30.306	32.451	643.652

utility, which can simulate network characteristics such as packet delay, loss, and duplication. By using NetEm, we eliminate the potential for uneven network speeds between the servers as there are several factors that could affect the time it takes a message to pass from one server to another.

Using a single server simplified considerably our experimental setting; however, this did not affect the accuracy of our experiments since we are only interested in relative slowdown of query answering for different packet delays.

9.2 Evaluation Protocol

A ping roundtrip between servers on a LAN typically takes less than 1 ms, while on a WAN it can take anywhere between 10 ms and 100 ms. Thus, to simulate such scenarios, we ran with 0 ms delay then reran our queries while delaying packets on the loopback interface by 1 ms, 10 ms, and 100 ms.

9.3 Experiment Results

When the number of partial answers is similar on all partitioning strategies, a network slowdown affects all approaches in roughly the same way. Our measurements have confirmed this observation so, for the sake of the discussion, in Table 9.1 we show our results only four queries T1, T7, N2 and N3 of LUBM. These cases are interesting because the numbers of partial answers vary by orders of magnitude depending on the partitioning strategy.

9.4 Discussion

The effect of slowing down the network is mostly felt when there is high communication between the servers. For most of the queries, the amount of communication needed between the servers is almost the same across the different approaches and as a result, there isn't much of a difference in query evaluation times as they are all equally affected.

From Table 9.1, it is evident that reducing communication becomes more important as the network slows down. In particular, C-L and C-T are hardly affected by any network slowdown, which is due to the fact that very few partial bindings are needed to answer these queries. As a result, the overall evaluation will still happen faster. For all the queries displayed in Table 9.1, HSH is the most affected in with query times going from 82,330ms to 3,679,016ms for query N2 which is orders of magnitude more than any of the other approaches although all the approaches increase in evaluation time as well.

A perfect example is query T1, with no delay, all approaches evaluate the query and complete in similar times. However, when we increase the delay, we see that C-L, C-T and PRN are not affected much by the increase in delay. With the highest delay, we see C-T, C-L and PRN increase in evaluation time by less than 25% with a delay of 100ms but HSH increases by 3,805%. This is due to the fact that HSH sends 22.67 million partial bindings messages with PRN sending 72380 messages whilst C-L sent only 2090 messages and and C-T sent 1780 as is shown in Table 7.1.

In contrast, communication overhead is orders of magnitude larger with HSH, which is reflected in query answering times: a packet delay of 100 ms incurs a slowdown of 8,330% on query N2. For PRN, the numbers of partial answers are larger than C-L and C-T, but much smaller than HSH, which is reflected in query running times: network slowdown affects PRN more than C-L and C-T, but much less than HSH.

9.5 Conclusion

Our results show that reducing communication can be important to improving query answering times. However, small reductions, such as between PRN and C-L or C-T, are unlikely to significantly improve query answering times; moreover, small improvements should not be made at the expense of unevenly distributed workloads. In contrast, a significant amount of network communication can become a dominant factor determining the performance of query evaluation on very slow networks. However, given the advances in technology, slow networks are uncommon in practice especially for systems hosting web-scale datasets. As such, we do not envisage network communication would be a large factor to worry about when developing an effective partitioning strategy.

10

Conclusion and Outlook

10.1 Conclusion

In this thesis we present novel advancements in partitioning for large RDF datasets and a thorough examination of the effects of partitioning on query answering. We identified certain misconceptions in the literature that have motivated several partitioning strategies such as the importance of minimizing communication between servers for quicker query evaluation times.

We present two new techniques for partitioning RDF data based on the Louvain graph community detection algorithm. The main motivation behind our work was producing partitions that require less distributed join processing so that communication between servers is reduced. Both techniques succeed in reducing communication during distributed join processing, and we compared them empirically with well-known data partitioning techniques.

While communication is commonly assumed to be the main source of overhead, we show that the distribution of workload across servers in a cluster can also have a significant impact on the performance of query answering in distributed RDF systems. We initially developed the variant with tight community allocation (C-T) that produced strong partitions where the resource overlap was minimised.

Partitioning by communities drastically reduced communication between the servers during query evaluation, however, during evaluation between C-T, PRN and HSH, we observe that the approaches with orders of magnitude more communication than our approach were evaluating some of the queries faster. In fact, for some queries, C-T was worst performing despite the reduction in communication. This suggests that there are other side effects of partitioning besides communication that affect the query evaluation process.

We were quite surprised by these results: it is commonly assumed in the literature that network communication is the main factor for the performance of distributed join processing. In order to understand exactly what is happening, we do a thorough analysis of different aspects of the query evaluation process.

During the query evaluation, we measure the number of partial messages sent per server, and from the results, we are able to have an indication of which servers are busy whilst others are idle. We notice that for the approaches that provide very strong partitions with reduced communication between the servers such as C-T, C-L and PRN, the reduction in communication is usually because most of the triples required to satisfy the queries reside on a few servers. As a result, the query evaluation is slowed down by the server with the heaviest workload.

To further refine our findings, we measured the atom matches made per server for each of the approaches. This gave us a true representation of the server workload because each query is broken into atoms which are run on the system and partial answer messages are only sent if the processing cannot continue on the current server.

The matches per server further confirmed our suspicions, for example, for query L4, C-T had some servers that did absolutely no work to help satisfy the query with zero matches made whilst for the same query we had one server making 1.58M

matches alone. On the other hand, HSH had an average of 165k matches per server. Despite the contrast in the messages sent, HSH finishes faster than all the approaches in 0.289s whilst C-T finishes in 1.576s.

Our results suggest that communication overhead is only one side of the equation: on fast networks and when communication overhead is not excessive, ensuring that servers in the cluster are loaded evenly may actually be much more important. We use this as the main motivation for the C-L variant of our technique, which also partitions the dataset by community but we limit the size of the communities as well as distribute communities across servers randomly without trying to minimise the resource overlap as is the case with C-T. This partitioning variant aims to strike a balance between reducing communication and distributing workload.

The loose allocation variant C-L and HSH perform better in running queries on WatDiv but C-T and PRN performs better on LUBM. This is due to the dataset types. WatDiv is more dense with extremely strong communities whilst LUBM is easier to partition and sparse. This indicates that the performance during query evaluation is also dependent on the dataset as well.

To make this study self contained, we measured the effects of query evaluation if the network is slowed down. We slowed the network by 1ms, 10ms, and 100ms. We find that with partitioning strategies that have minimal communication such as C-T, C-L and PRN, the network delay has negligible results on the query evaluation times. However, for HSH, the delays are felt with the evaluation time increasing by as a much as 8,330% where $delay = 100ms$ for query N2.

It is important to note however that networks are faster and a delay of 100ms is not common in practice for a message to be sent. As a result, given the current state of the art in computers, processor and networks, we classify network speed as the least probable factor to slow down communication as a slow network is highly unlikely.

In conclusion, there are several factors that affect query evaluation over a distributed architecture. In the literature, a lot of emphasis is put on reducing the amount of communication and that this would lead to faster query evaluation

times. We have seen that this is not always the case. Due to that, no approach can claim to be the best partitioning for every dataset.

We evaluated the different partitioning approaches and observed the behaviour during query evaluation and have come to several observations. Firstly, there is no one-size-fits-all partitioning strategy for RDF. The dataset structure, the partitioning strategy, the types of queries all play a role in the performance of the system. Secondly, contrary to popular belief, the most important aspect to optimise is server workload balance and not communication reduction. Third, when workload is evenly distributed among the servers, the race to completion is based on the amount of communication by the approach. Finally, network communication, whilst it can be a bottleneck, given the current state of the art, it is the least likely to affect query evaluation times.

10.2 Future Work

In future, we will refine our data partitioning technique to further improve the balance between network communication and server workload and thus ensure consistent performance on a wide range of datasets and query loads. A particular challenge is to develop techniques that can partition an RDF graph in a *streaming* fashion—that is, by reading the input graph sequentially and keeping only small subsets of the graph in memory at once.

As has been seen in the evaluation that several factors determine the performance during query evaluation, we would also like to create a dynamic partitioning algorithm for RDF. The dynamic partitioning engine should be able, given the dataset, sample queries, and the number of servers, suggest the best partitioning strategy to be used.

Appendices

A

Queries Used in Evaluation

A.1 LUBM Queries

The Lehigh University Benchmark (LUBM) [27] benchmark comes with 14 queries; however, without taking the ontology into account, most of them do not return any results. Thus, we used the seven queries (T1–T7) by Zeng et al. [14] that compensate for the lack of reasoning, and three new, complex cyclic queries (N1–N3) by Potter et al. [25]. The prefixes used are below.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub:  <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
```

```
T1
SELECT ?X ?Y ?Z WHERE {
    ?X rdf:type ub:GraduateStudent .
```

```
    ?X ub:undergraduateDegreeFrom ?Y .
    ?X ub:memberOf ?Z .
    ?Z ub:subOrganizationOf ?Y .
    ?Z rdf:type ub:Department .
    ?Y rdf:type ub:University .
}

T2
SELECT ?X ?Y WHERE {
    ?X rdf:type ub:Course .
    ?X ub:name ?Y .
}

T3
SELECT ?X ?Y ?Z WHERE {
    ?Y rdf:type ub:University .
    ?X ub:undergraduateDegreeFrom ?Y .
    ?X rdf:type ub:UndergraduateStudent .
    ?X ub:memberOf ?Z .
    ?Z rdf:type ub:Department .
    ?Z ub:subOrganizationOf ?Y .
}

T4
SELECT ?X ?Y1 ?Y2 ?Y3 WHERE {
    ?X rdf:type ub:Professor .
    ?X ub:worksFor <http://www.Department0.University0.edu> .
    ?X ub:name ?Y1 .
    ?X ub:emailAddress ?Y2 .
    ?X ub:telephone ?Y3
```

```
}

T5
SELECT ?X WHERE {
    ?X ub:subOrganizationOf <http://www.Department0.University0.edu> .
    ?X rdf:type ub:ResearchGroup .
}

T6
SELECT ?X ?Y WHERE {
    ?Y ub:subOrganizationOf <http://www.University0.edu> .
    ?Y rdf:type ub:Department .
    ?X ub:worksFor ?Y .
    ?X rdf:type ub:FullProfessor .
}

T7
SELECT ?X ?Y ?Z WHERE {
    ?Y rdf:type ub:FullProfessor .
    ?Y ub:teacherOf ?Z .
    ?Z rdf:type ub:Course .
    ?X ub:advisor ?Y .
    ?X ub:takesCourse ?Z .
    ?X rdf:type ub:UndergraduateStudent .
}

N1
SELECT ?PROF ?DEPT ?STUD ?UNIV WHERE {
    ?DEPT ub:subOrganizationOf ?UNIV .
    ?PROF ub:worksFor ?DEPT .
```

```
    ?STUD ub:advisor ?PROF .
    ?STUD ub:undergraduateDegreeFrom ?UNIV .
}

N2
SELECT ?S1 ?C1 ?P1 ?C2 ?S2 ?C3 WHERE {
    ?S2 ub:teachingAssistantOf ?C2 .
    ?P1 ub:teacherOf ?C2 .
    ?P1 ub:teacherOf ?C1 .
    ?S2 ub:takesCourse ?C3 .
    ?S1 ub:takesCourse ?C3 .
    ?S1 ub:takesCourse ?C1 .
}

N3
SELECT DISTINCT ?S1 WHERE {
    ?S2 ub:teachingAssistantOf ?C2 .
    ?S2 ub:takesCourse ?C1 .
    ?P1 ub:teacherOf ?C1 .
    ?P1 ub:teacherOf ?C2 .
    ?S2 ub:takesCourse ?C3 .
    ?S1 ub:takesCourse ?C3 .
    ?S1 ub:takesCourse ?C1 .
}
```

A.2 Watdiv Queries

WatDiv provides 20 query templates, which have been divided into four classes: star (S), linear (L), snowflake (F), and complex (C) queries. Since star queries contain only subject–subject joins, they do not incur any communication on either

of the data partitioning approaches we consider and are thus not interesting in our work. Consequently, we considered only the 13 query templates consisting of linear, snowflake, and complex queries. These templates contain a parameter that is usually replaced with a resource from the graph; however, queries obtained in this way tend to be localised and thus do not require much distributed processing. To obtain more complex and interesting queries, we replaced each parameter in a query template with a fresh variable. The prefixes used are below.

```
PREFIX wsdbm: <http://db.uwaterloo.ca/~galuc/wsdbm/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sorg:  <http://schema.org/>
```

```
C1
SELECT ?v0 ?v4 ?v6 ?v7 WHERE {
    ?v0 <http://schema.org/caption> ?v1 .
    ?v0 <http://schema.org/contentRating> ?v3 .
    ?v0 <http://schema.org/text> ?v2 .
    ?v0 <http://purl.org/stuff/rev#hasReview> ?v4 .
    ?v4 <http://purl.org/stuff/rev#title> ?v5 .
    ?v4 <http://purl.org/stuff/rev#reviewer> ?v6 .
    ?v7 <http://schema.org/actor> ?v6 .
    ?v7 <http://schema.org/language> ?v8 .
}
```

```
C2
SELECT ?v0 ?v3 ?v4 ?v8 WHERE {
    ?v2 <http://schema.org/eligibleRegion> ?v10 .
    ?v0 <http://purl.org/goodrelations/offers> ?v2 .
    ?v0 <http://schema.org/legalName> ?v1 .
    ?v2 <http://purl.org/goodrelations/includes> ?v3 .
```

```
    ?v3 <http://purl.org/stuff/rev#hasReview> ?v8 .
    ?v8 <http://purl.org/stuff/rev#totalVotes> ?v9 .
    ?v7 wsdbm:purchaseFor ?v3 .
    ?v4 wsdbm:makesPurchase ?v7 .
    ?v4 <http://xmlns.com/foaf/homepage> ?v6 .
    ?v4 <http://schema.org/jobTitle> ?v5 .
}
```

C3

```
SELECT ?v0 WHERE {
    ?v0 <http://purl.org/dc/terms/Location> ?v3 .
    ?v0 wsdbm:likes ?v1 .
    ?v0 <http://xmlns.com/foaf/age> ?v4 .
    ?v0 wsdbm:gender ?v5 .
    ?v0 <http://xmlns.com/foaf/givenName> ?v6 .
    ?v0 wsdbm:friendOf ?v2 .
}
```

F1

```
SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE {
    ?v0 <http://ogp.me/ns#tag> ?v1 .
    ?v0 rdf:type ?v2 .
    ?v3 wsdbm:hasGenre ?v0 .
    ?v3 <http://schema.org/trailer> ?v4 .
    ?v3 <http://schema.org/keywords> ?v5 .
    ?v3 rdf:type wsdbm:ProductCategory2 .
}
```

F2

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 WHERE {
```

```
    ?v0 wsdbm:hasGenre ?v8 .
    ?v0 <http://xmlns.com/foaf/homepage> ?v1 .
    ?v0 <http://ogp.me/ns#title> ?v2 .
    ?v0 rdf:type ?v3 .
    ?v0 <http://schema.org/caption> ?v4 .
    ?v0 <http://schema.org/description> ?v5 .
    ?v1 wsdbm:hits ?v7 .
    ?v1 <http://schema.org/url> ?v6 .
}
```

F3

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 WHERE {
    ?v0 wsdbm:hasGenre ?v3 .
    ?v0 <http://schema.org/contentRating> ?v1 .
    ?v0 <http://schema.org/contentSize> ?v2 .
    ?v5 wsdbm:purchaseFor ?v0 .
    ?v5 wsdbm:purchaseDate ?v6 .
    ?v4 wsdbm:makesPurchase ?v5 .
}
```

F4

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
    ?v0 <http://ogp.me/ns#tag> ?v3 .
    ?v0 <http://xmlns.com/foaf/homepage> ?v1 .
    ?v0 <http://schema.org/description> ?v4 .
    ?v0 <http://schema.org/contentSize> ?v8 .
    ?v1 <http://schema.org/language> wsdbm:Language0 .
    ?v1 <http://schema.org/url> ?v5 .
    ?v1 wsdbm:hits ?v6 .
    ?v2 <http://purl.org/goodrelations/includes> ?v0 .
```

```
    ?v7 wsdbm:likes ?v0 .
}

F5
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
    ?v2 <http://purl.org/goodrelations/offers> ?v0 .
    ?v0 <http://purl.org/goodrelations/includes> ?v1 .
    ?v0 <http://purl.org/goodrelations/price> ?v3 .
    ?v0 <http://purl.org/goodrelations/validThrough> ?v4 .
    ?v1 <http://ogp.me/ns#title> ?v5 .
    ?v1 rdf:type ?v6 .
}

L1
SELECT ?v0 ?v2 ?v3 WHERE {
    ?v0 wsdbm:subscribes ?v1 .
    ?v0 wsdbm:likes ?v2 .
    ?v2 <http://schema.org/caption> ?v3 .
}

L2
SELECT ?v1 ?v2 WHERE {
    ?v0 <http://www.geonames.org/ontology#parentCountry> ?v1 .
    ?v2 <http://schema.org/nationality> ?v1 .
    ?v2 wsdbm:likes wsdbm:Product0 .
}

L3
SELECT ?v0 ?v1 WHERE {
    ?v0 wsdbm:subscribes ?v2 .
```

```
    ?v0 wsdbm:likes ?v1 .
}
```

L4

```
SELECT ?v0 ?v2 WHERE {
    ?v0 <http://ogp.me/ns#tag> ?v1 .
    ?v0 <http://schema.org/caption> ?v2 .
}
```

L5

```
SELECT ?v0 ?v1 ?v3 WHERE {
    ?v0 sorg:jobTitle ?v1 .
    ?v2 <http://www.geonames.org/ontology#parentCountry> ?v3 .
    ?v0 sorg:nationality ?v3 .
}
```

References

[1] Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J Carroll, and Brian McBride. "RDF 1.1 concepts and abstract syntax". In: *W3C recommendation* 25.02 (2014).

[2] Dan Brickley, Ramanathan V Guha, and Brian McBride. "RDF Schema 1.1". In: *W3C recommendation* 25 (2014), pp. 2004–2014.

[3] Dan Brickley, Ramanathan V Guha, and Brian McBride. "RDF Schema 1.1". In: *W3C recommendation* 25 (2014), pp. 2004–2014.

[4] World Wide Web Consortium et al. "SPARQL 1.1 overview". In: (2013).

[5] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and complexity of SPARQL". In: *ACM Transactions on Database Systems (TODS)* 34.3 (2009), pp. 1–45.

[6] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. "RDF data storage and query processing schemes: A survey". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–36.

[7] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. "Jena: Implementing the Semantic Web Recommendations". In: *WWW*. 2004, pp. 74–83.

[8] J. Broekstra, A. Kampman, and F. van Harmelen. "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema". In: *ISWC*. 2002, pp. 54–68.

[9] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. "The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases". In: *SemWeb*. 2001.

[10] Thomas Neumann and Gerhard Weikum. “The RDF-3X engine for scalable management of RDF data”. In: *VLDB Journal* 19.1 (2010), pp. 91–113.

[11] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefam Decker. “YARS2: A Federated Repository for Querying Graph Structured Data from the Web”. In: *ISWC*. 2007, pp. 211–224.

[12] Steve Harris, Nick Lamb, and Nigel Shadbolt. “4store: The Design and Implementation of a Clustered RDF Store”. In: *SSWS*. Vol. 517. CEUR Workshop Proceedings. 2009, pp. 94–109.

[13] Jiewen Huang, Daniel J Abadi, and Kun Ren. “Scalable SPARQL Querying of Large RDF Graphs”. In: *PVLDB* 4.11 (2011), pp. 1123–1134.

[14] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin hao, and Zhongyuan Wang. “A Distributed Graph Engine for Web Scale RDF Data”. In: *PVLDB* 6.4 (2013), pp. 265–276.

[15] Kurt Rohloff and Richard E. Schantz. “Clause-Iteration with MapReduce to Scalably Query Data Graphs in the SHARD Graph-Store”. In: *DIDC*. 2011, pp. 35–44.

[16] Kisung Lee and Ling Liu. “Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning”. In: *PVLDB* 6.14 (2013), pp. 1894–1905.

[17] Luis Galárraga, Katja Hose, and Ralf Schenkel. “Partout: a distributed engine for efficient RDF processing”. In: *WWW*. 2014, pp. 267–268.

[18] 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”. In: *VLDB Journal* 25.3 (2016), pp. 355–380.

[19] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. “TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing”. In: *SIGMOD*. 2014, pp. 289–300.

[20] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Hai Jin, and Ling Liu. "SemStore: A Semantic-Preserving Distributed RDF Triple Store". In: *CIKM*. 2014, pp. 509–518.

[21] 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". In: *PVLDB* 8.6 (2015), pp. 654–665.

[22] Katja Hose and Ralf Schenkel. "WARP: Workload-aware replication and partitioning for RDF". In: *Workshop at ICDE*. 2013, pp. 1–6.

[23] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (1998), pp. 359–392.

[24] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. "A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data". In: *PVLDB* 10.13 (2017), pp. 2049–2060.

[25] Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks. "Dynamic Data Exchange in Distributed RDF Stores". In: *IEEE TKDE* 30.12 (2018), pp. 2312–2325.

[26] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. "Fast unfolding of communities in large networks". In: *J. Stat. Mech.* 2008.10 (2008), P10008.

[27] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems". In: *J. Web Semantics* 3.2 (2005), pp. 158–182.

[28] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. "Diversified Stress Testing of RDF Data Management Systems". In: *ISWC*. 2014, pp. 197–212.

[29] Yannis E Ioannidis and Younkyung Cha Kang. "Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization". In: *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*. 1991, pp. 168–177.

[30] Mario Arias Gallego, Javier D Fernández, Miguel A Martìnez-Prieto, and Pablo de la Fuente. “An Empirical Study of Real-World SPARQL Queries”. In: *CoRR* abs/1103.5043 (2011).

[31] Jianling Sun and Qiang Jin. “Scalable rdf store based on hbase and mapreduce”. In: *2010 3rd international conference on advanced computer theory and engineering (ICACTE)*. Vol. 1. IEEE. 2010, pp. V1–633.

[32] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. “H_2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs”. In: *Big Data*. 2013, pp. 255–263.

[33] Raghava Mutharaju, Sherif Sakr, Alessandra Sala, and Pascal Hitzler. “D-SPARQ: distributed, scalable and efficient RDF query engine”. In: *ISWC (Posters & Demos)*. Vol. 1035. CEUR Workshop Proceedings. 2013, pp. 261–264.

[34] Goetz Graefe and Diane L. Davison. “Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution”. In: *IEEE Trans. Software Eng.* 19.8 (1990), pp. 749–764.

[35] David C Faye, Olivier Cure, and Guillaume Blin. “A survey of RDF storage approaches”. In: *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées* 15 (2012), pp. 11–35.

[36] Zoi Kaoudi and Ioana Manolescu. “RDF in the clouds: a survey”. In: *The VLDB Journal* 24.1 (2015), pp. 67–91.

[37] Zoi Kaoudi, Ioana Manolescu, and Stamatis Zampetakis. “Cloud-Based RDF Data Management”. In: *Synthesis Lectures on Data Management* 15.1 (2020), pp. 1–103.

[38] Bin Shao, Haixun Wang, and Yatao Li. “Trinity: A distributed graph engine on a memory cloud”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013, pp. 505–516.

[39] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. "H 2 RDF+: High-performance distributed joins over large-scale RDF graphs". In: *2013 IEEE International conference on big data*. IEEE. 2013, pp. 255–263.

[40] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. "H2RDF: adaptive query processing on RDF data in the cloud." In: *Proceedings of the 21st International Conference on World Wide Web*. 2012, pp. 397–400.

[41] Deepak Vohra. *Apache HBase Primer*. Vol. 151. Springer, 2016.

[42] C. Weiss, P. Karras, and A. Bernstein. "Hexastore: Sextuple Indexing for Semantic Web Data Management". In: *VLDB* 1.1 (2008), pp. 1008–1019.

[43] X. Zhang, L. Chen, Y. Tong, and M. Wang. "EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud". In: *ICDE*. 2013, pp. 565–576.

[44] Mohammad F. Husain, James P. McGlothlin, Mohammad M. Masud, Latifur R. Khan, and Bhavani M. Thuraisingham. "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing". In: *IEEE TKDE* 23.9 (2011), pp. 1312–1327.

[45] Olaf Görlitz and Steffen Staab. "Splendid: SPARQL endpoint federation exploiting void descriptions". In: *Consuming Linked Data*. 2011, pp. 13–24.

[46] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. "Processing SPARQL queries over distributed RDF graphs". In: *VLDB Journal* 25.2 (2016), pp. 243–268.

[47] Lei Zou, M Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. "gStore: a graph-based SPARQL query engine". In: *The VLDB journal* 23.4 (2014), pp. 565–590.

[48] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. "TripleBit: a fast and compact system for large scale RDF data". In: *Proceedings of the VLDB Endowment* 6.7 (2013), pp. 517–528.

[49] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. "S2RDF: RDF Querying with SPARQL on Spark". In: *CoRR* abs/1512.07021 (2015).

[50] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[51] Alexander Schätzle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg Lausen. "Sempala: Interactive SPARQL Query Processing on Hadoop". In: *ISWC*. 2014, pp. 164–179.

[52] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. "An intermediate algebra for optimizing RDF graph pattern matching on MapReduce". In: *Extended semantic web conference*. Springer. 2011, pp. 46–61.

[53] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. "Scan-sharing for optimizing rdf graph pattern matching on mapreduce". In: *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE. 2012, pp. 139–146.

[54] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. "PigSPARQL: A SPARQL Query Processing Baseline for Big Data". In: *ISWC (Poster)*. 2013, pp. 241–244.

[55] M. E. J. Newman. "Analysis of weighted networks". In: *Phys. Rev. E* 70 (5 2004), p. 056131.

[56] Kitty Meeks and Fiona Skerman. "The parameterised complexity of computing the maximum modularity of a graph". In: *Algorithmica* (2019), pp. 1–26.

[57] E. A. Leicht and M. E. J. Newman. "Community Structure in Directed Networks". In: *Phys. Rev. Lett.* 100 (11 2008), p. 118703.

[58] Sang Gyu Kwak and Jong Hae Kim. "Central limit theorem: the cornerstone of modern statistics". In: *Korean journal of anesthesiology* 70.2 (2017), p. 144.

[59] Fredah Banda and Boris Motik. "Community-Based RDF Graph Partitioning". In: *Proc. SSWS*. 2020.