Query Answering in Distributed RDF Databases

Anthony Potter

St. Cross

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2017

Acknowledgements

I would like to express my most sincere thanks to my supervisors Prof. Boris Motik and Prof. Ian Horrocks, without whom this thesis would not have been possible. They gave incredible guidance and support throughout my studies. I would also like to thank Dr. Yavor Nenov, who was my supervisor in all but title. Our many discussions and whiteboard sessions were invaluable.

My thanks goes out to all those I have met during my time at Oxford, the members of New College, St. Cross College, and the Department of Computer Science. They are the people that make Oxford such a vibrant city to live and learn in.

I'm eternally grateful to all my family and friends who helped me along the way. To my parents for always being there, to Gary, Emma, and Barkley for their care and support, and to Steve and Piet for always throwing up in the most unexpected places.

Finally, the person who has been by my side and closest to my heart the entire journey, Ingrid. From Noodle Nation to V. Sattui, India to California, you have been there as my cheerleader and my inspiration. Thank you for the true and endless happiness you have brought me.

Abstract

To simplify data integration and exchange, modern applications often represent their data using the Resource Description Framework (RDF). As the amount of the available data keeps increasing, many RDF datasets cannot be processed using centralised RDF stores. A common solution is to distribute RDF data in a cluster of shared-nothing servers, and to query the data using a distributed query algorithm. Existing approaches typically use a variant of the *data exchange operator* to shuffle partial query answers between servers and thus ensure that every query answer is produced. Decisions as to when and where to shuffle the data are usually made *statically*—that is, at query compile time. In this thesis, we argue that such approaches can miss opportunities for local computation and thus incur considerable overheads. Moreover, we present a novel distributed query evaluation algorithm for RDF based on *dynamic data exchange*, where all computation that can be done locally is guaranteed to be performed on a single server. Our approach can successfully process any query even if the memory available at each server is bounded, and we argue that this is critical in distributed systems where intermediate results can easily exceed the capacity of each server. We also present a new query planning approach that balances the cost of communication against the cost of local processing at each server, as well as a new approach to partitioning RDF data that aims to increase locality in each server. We have implemented our approach in the well-known RDFox data store, and our empirical evaluation suggests that our techniques can outperform the state of the art by orders of magnitude in terms of query evaluation times, network communication, and memory use.

Publications

Anthony Potter, Boris Motik, and Ian Horrocks. “Querying Distributed RDF Graphs: The Effects of Partitioning”. In: Proceedings of the 10th International Workshop on Scalable Semantic Web Knowledge Base Systems. 2014.

Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks. “Distributed RDF Query Answering with Dynamic Data Exchange”. In: International Semantic Web Conference. Springer. 2016, pp. 480–497.

Nicholas Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. “PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine”. In: Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems. GRADES’17. Chicago, IL, USA: ACM, 2017, 7:1–7:6.

Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks. "Dynamic Data Exchange in Distributed RDF Stores". In IEEE Transactions on Knowledge and Data Engineering. 2018.

Contents

List of Figures

List of Tables

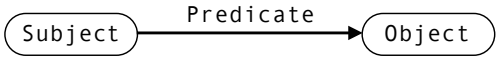
1 Introduction

The Semantic Web is intended to extend the current web to be machine readable by ensuring the information that it contains has a well defined meaning and is available in a standard format. Importantly, access to the data should be accompanied by access to the *relationships* between data. The collection of data on the Web and their relationships is referred to as Linked Data. In order to achieve Linked Data, common formats, standards, and technologies are required to enable the integration, use, and processing of the data. The Resource Description Framework (RDF) [42] has been recommended by the World Wide Web Consortium (W3C) as the standard format for representing information in the Web.

1.1 Resource Description Framework

RDF is a popular graph-like data model designed to simplify data integration and data exchange. It is unstructured with no schema, allowing the data to be self-describing. The core element of RDF is a triple, consisting of a subject, a predicate, and an object. A set of triples is called an RDF graph, as it can be represented as a labelled, directed graph where the subjects and objects of the triples are the vertices and the predicates are the edges that connect them, as shown in Figure 1.1.

Figure 1.1: Graphical view of an RDF triple

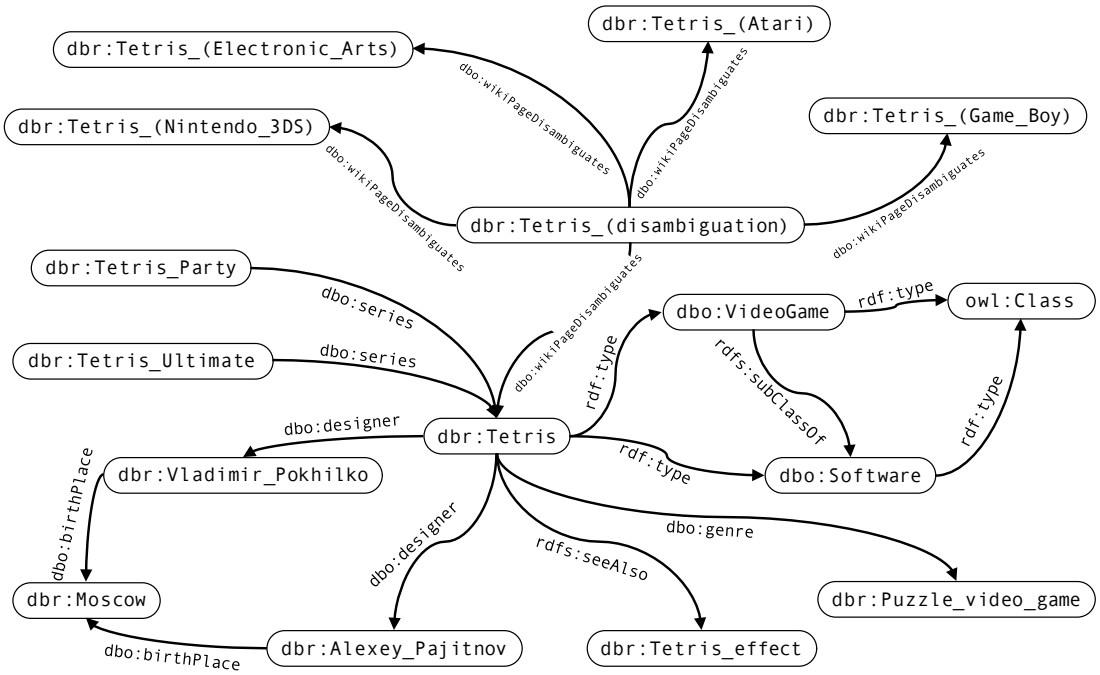


Any object in the world, or universe of discourse, can be represented by a resource, including anything from people or blobs of data to abstract concepts. These resources are represented by either Internationlized Resource Identifiers (IRIs) or literals. IRIs act as globally unique identifiers for resources, whereas literals describe a resource's literal value and are used for values such as strings, integers, and dates. Additionally, blank nodes can be used to represent incomplete data, in terms of the existence of a resource with the given relationships, without explicitly naming it. The assertion of an RDF triple has a specific meaning which says that the resource represented by the subject has some relationship, described by the predicate, with the resource represented by the object. An RDF graph is a set of RDF triples.

An example RDF graph can be seen in Figure 1.2, which is an extract from the DBPedia dataset [7] relating to Tetris. The resources represent a wide range of entities, from people such as *dbr*:*Vladimir_Pokhilko*, phenomenon such as *dbr*:*Tetris_effect*, and classes such as *dbo*:*VideoGame*. The triples describe the relationships between these entities, for example the triple ⟨*dbr*:*Vladimir_Pokhilko*, *dbo*:*birthPlace*, *dbr*:*Moscow*⟩ says that Vladimir Pokhilko was born in Moscow.

The example in Figure 1.2 also illustrates an important property of RDF with the inclusion of schema information in the form of triples, such as ⟨*dbo*:*VideoGame*, *rdfs*:*subClassOf*, *dbo*:*Software*⟩ describing a subclass relation. This is in contrast to traditional databases that separate the instance data from the schema information [2]. Many typical concepts have been standardised by the W3C through RDF Schema (RDFS) [9]. RDFS is a semantic extension of RDF that provides a facility to standardise vocabularies across datasets so that their meaning can be interpreted in the intended way. It introduces a set of classes and properties, along with associated semantics.

Figure 1.2: Sample RDF graph from DBPedia



1.2 SPARQL

A fundamental component of any database is the ability to query the data. The standard language for querying RDF data, as set out by the W3C, is SPARQL [26]. The basic building block of a typical SPARQL query is the *basic graph pattern* (BGP); queries which contain only BGPs are analogous to Select-Project-Join database queries. BGPs are conjunctions of triple patterns, which are extensions of triples to include variables. As BGPs are conjunctions of triple patterns, they are sometimes referred to as RDF conjunctive queries. The syntax is shown in (1.1), where b_i are BGPs and v_i are variables which must also occur in some b_j.

$$\texttt{SELECT}\ ?v_1 \ldots ?v_n\ \texttt{WHERE}\ \{\ b_1 \ldots b_m\ \} \tag{1.1}$$

The `SELECT` clause identifies all the answer variables (i.e. those that will appear in the results set) and the `WHERE` clause provides the BGPs to be matched. SPARQL does not use the join operator from relational algebra, but instead joins are expressed

by repeating variables or resources. As an example, suppose we want to query the graph in Figure 1.2. Equation (1.2) shows a SPARQL query to find all of the designers of Tetris, along with where they were born.

```
SELECT ?x ?y WHERE {
        dbr:Tetris dbo:designer ?x .
        ?x dbo:birthPlace ?y }
```
(1.2)

There is a join between both triple patterns, with the repetition of the x variable. The result of executing such a query is shown in the table below.

x	y
dbr:Vladimir_Pokhilko	dbr:Moscow
dbr:Alexey_Pajitnov	dbr:Moscow

SPARQL also provides many other query features common to many database query languages, such as `OPTIONAL`, `FILTER`, `UNION`, `ORDER BY`, and `LIMIT`.

1.3 Database Management Systems

The two main functions on RDF data this thesis concerns are storage and querying, which require a database management system (DBMS). Approaches for RDF-based DMBSs can be classified as either *native* or *non-native*; native systems are designed specifically for the management of RDF data, whereas non-native systems utilise existing DBMSs, such as relational DBMSs or key-value stores. Furthermore, a distinction can be made at the physical storage level, where the data is stored persistently on disk, or loaded into main memory. Increasingly, performance-critical applications are using memory-based approaches as it offers significant performance benefits in terms of data access.

As the number of RDF data sources keeps increasing, RDF datasets used in practice are becoming too large to be processed on a single server. This problem is

exacerbated by those performance-critical applications that use in-memory RDF stores, such as financial services and fraud detection [35], whose capacity is limited by the cost of RAM. Moreover, Linked Data applications may integrate several large datasets that cannot be jointly processed even using disk-based systems.

To address such use cases, approaches to storing and querying RDF data in a cluster of shared-nothing servers have been developed [28, 30–32, 43, 49, 50, 53, 55–57, 71, 74]. Each approach typically comprises a query answering algorithm and a data partitioning strategy designed to address three main challenges. First, to compute a join, potentially expensive network communication may be required to bring triples participating in a join to one server. Second, synchronisation between servers can be costly and may considerably reduce the potential for parallel processing. Third, the intermediate results produced during join evaluation often grow with the overall data size and so they may easily exceed the capacity of individual servers.

The Volcano database system [25] was one of the first to address these challenges using a *data exchange* (or *shuffle*) *operator*, which encapsulates the communication between query execution processes. Such operators are added into query plans as needed to ensure that all other operators in the plan receive all the relevant data. Decisions about data exchange are usually made *statically* (i.e., during query planning) and are based on the properties of data partitioning. Moreover, a strategy can replicate data across servers in order to reduce the need for data exchange. Servers often process such plans synchronously (i.e., they progress through the operators in the plan in lockstep).

As we discuss in detail in Chapter 3, all distributed RDF stores we are aware of can be seen as using a variant of the above approach, each striking a different trade-off between data replication and data exchange. We also argue that static decisions about data exchange often incur a communication cost even when triples participating in a join are colocated and so no communication is needed. Network

communication is one of the main bottlenecks in distributed RDF stores [32], so reducing and/or eliminating it has a direct effect on the performance and scalability of distributed systems.

1.4 Data Partitioning

Distributed systems introduce further challenges for data storage, not only in the data structures used, but particularly on which server to store the data. Data partitioning groups data and associates it with a particular location, which can be, for example, a device, component, file, or server. Moreover, it also encapsulates the duplication of data across multiple locations, which can provide redundancy for fault-tolerance or improve system performance at the cost of storage overhead.

Data partitioning schemes can have a significant impact on many aspects of the performance of query answering, such as the response time, how evenly the work done is balanced across servers, and the amount of network communication that is necessary. Additionally, the amount of space and time needed to perform different partitioning schemes can be an important consideration depending on the application or amount of resources available. For example, partitioning schemes based on hashing are typically fast with little memory requirements, whereas graph partitioning schemes can require large amounts of memory and take considerable time to compute.

With the data exchange operator seeing such wide use, and network communication often being a bottleneck in distributed systems, a major goal for the data partitioning schemes used in recent systems has been to reduce the number of data exchange operators used in evaluating typical queries. This is done through the data partitioning schemes providing strong guarantees on where the data is placed. For example, if it guarantees that all triples with the same subject will be placed on the same server, any subject-subject join can be computed across a cluster without the use of a data exchange operator and thus without any network communication.

In order to provide strong enough guarantees to have an appreciable effect on performance, many approaches use aggressive duplication, which can limit scalability.

1.5 Query Planning

SPARQL is a declarative query language, and hence it does not specify *how* a query is evaluated. The DBMS is responsible for translating a query into a query evaluation plan, which is a specification of the operations, and their order, necessary to compute the answers to a query. Typically, an operator takes as input one or more streams of data, manipulates that data in some way, and then outputs the result in one or more streams. For example, the selection operator takes a stream of data and outputs the elements of the input data that satisfy some condition.

The query execution plan is often critical for performance, which can be illustrated quite simply. Consider a join between two sets of indexed data, R and S, where R has size 1 and S has size n. One strategy could be a sequential scan of R, using an index lookup on S to find a possible join, resulting in 1 index lookup. Another could be a sequential scan of S, using an index lookup on R, resulting in n index lookups. This shows the performance of even a single join can depend greatly on the query plan.

Query planners, or optimisers, are used to find a query plan that minimises the cost of executing a query. They typically comprise of three parts: (i) a *query cardinality estimator* that uses statistics about the data to estimate the cardinality, i.e. the number of answers, of a query, (ii) a *cost model* that combines these estimates and other knowledge of the system into a numeric measure of the cost of executing a query, and (iii) a *query planning algorithm* that attempts to find the lowest cost plan according to the cost model. Query cardinality estimation has been extensively studied and is done by summarising a database using synopses. There are many ways to summarise a database, such as with multidimensional

histograms [3, 10], or graphical models [22]. As the cost of a query plan is a function of query cardinalities, the ability of a query planner to identify efficient plans is critically dependent on the accuracy of the query cardinality estimator. Another key challenge for a query planner is defining an accurate cost model. This is particularly challenging in a distributed environment, as servers work in parallel and the additional cost of network communication must be modeled.

1.6 Contributions

In this thesis we present several novel algorithms and techniques in the domain of query answering in distributed RDF databases, as well as a prototypical implementation to evaluate their performance. We summarise those contributions below.

In Chapter 4, we present an approach to distributed query answering that uses a novel application of partial evaluation to distributed RDF databases. This approach guarantees that all answers to a query that can be computed from the data on a given server will be computed on that server. The aim of such a guarantee is to reduce the amount of network communication during query evaluation, which is often a bottleneck in distributed systems. Furthermore, we present a data partitioning strategy to complement this approach which is based on graph partitioning. This strategy aims to maximise the number of answers that can be computed on single servers, without the high storage overhead of data duplication that is typical in many modern systems. However, while effective in some cases, the approach has inherent limitations, the effects of which were clearly observable in an empirical evaluation.

In Chapter 5 we present a revised approach to distributed query answering that moves decisions about when and where to transfer data from the query planning stage to run time; we call this *dynamic data exchange*. This approach is the focus of the remaining chapters of the thesis. We argue that the common use of the data exchange operator misses opportunities for local computation and leads to

redundant network communication. We present a decentralised, asynchronous query evaluation algorithm that still guarantees that all local computation is done locally. Furthermore, we present optimisations that utilise this framework to improve efficiency and reduce the amount of network communication. We also present a novel, lightweight termination detection algorithm. Moreover, we show that the memory requirement for our query evaluation algorithm is bounded linearly by the number of atoms in the query, a property not found in existing systems. Lastly, we provide a full proof of correctness.

We then discuss query planning in a distributed system in Chapter 6. To attain good overall performance, a query plan must carefully balance the costs of local computation and network communication. To determine the former, query planners usually use a *cardinality estimator* that estimates the number of answers to any given query. To estimate the cost of network communication, we reduce the problem of determining the number of messages passed between the servers to the problem of estimating the cardinality of a specific query. Thus, we can estimate network communication cost by repurposing existing cardinality estimation techniques. We also discuss how to obtain a query plan that is optimal for the distributed system as a whole.

In Chapter 7 we present a novel RDF data partitioning method that aims to maximise data locality by using *graph partitioning* [40]—the task of dividing the vertices of a graph into sets while satisfying certain balancing constraints and minimising the number of edges between the sets. Graph partitioning has already been used for partitioning RDF data [28, 32], but these approaches duplicate data across servers to increase local processing. In contrast, our approach does not duplicate any data at all. Moreover, we use *weighted* graph partitioning to ensure that partitions are balanced in the number of triples, rather than the number of resources.

We have implemented our approach in the in-memory RDF management system RDFox [45], and have compared its performance with TriAD [28] (which has been shown to outperform other state of the art distributed RDF systems on a mix of data and query loads), S2RDF [57] (a Spark-based system), and PEDA [50] (a partial evaluation and assembly approach). In Chapter 8 we present the results of our evaluation using the LUBM [27] and WatDiv [4] benchmarks. We show that our approach outperforms the others in terms of query evaluation times, network communication, and memory usage; often by orders of magnitude.

2
Preliminaries

In this chapter, we recount the necessary background knowledge surrounding RDF, SPARQL, and conjunctive queries, as well as formally define the three main computational problems addressed in this thesis, namely; distributed query answering, data partitioning, and query planning. We first introduce some notation used throughout the thesis. For f a function, $\mathsf{dom}(f)$ is the domain of f, $\mathsf{rng}(f)$ is the range; for D a set, $f|_D$ is the restriction of f to $D \cap \mathsf{dom}(f)$; and if g is a function where $f(x) = g(x)$ for each $x \in \mathsf{dom}(f) \cap \mathsf{dom}(g)$, then $f \cup g$ is a function as well.

2.1 Resource Description Framework

RDF graphs are constructed from *resources*, which can be *IRI references*, *literals*, or *blank nodes*. Blank nodes have a special sematics, however, we do not use them in this work. A *triple* has the form $\langle t_s, t_p, t_o \rangle$ where t_s, t_p, and t_o are resources, and an *RDF graph* G is a finite set of triples. The *vocabulary* $\mathsf{voc}(G)$ of G is the set of all resources occurring in G. Let $\Pi = \{s, p, o\}$ be the set of *positions*. For $\pi \in \Pi$ a position, set $\mathsf{voc}_\pi(G)$ contains each resource occurring at position π of a triple in G.

2.2 SPARQL and Conjunctive Queries

SPARQL is an expressive language for querying RDF graphs; for example, the following SPARQL query retrieves all people who have a sister:

```
SELECT ?x WHERE {
       ?x rdf:type :Person .
       ?x :hasSister ?y }
```

SPARQL syntax is verbose, so we introduce a more compact notation to aid the presentation of the techniques and algorithms in this thesis. A *term* is a resource or a *variable*; in the rest of this thesis, variables are written using letters x, y, z, and w, which are sometimes indexed. An *atom* (aka *triple pattern*) A is an expression of the form $\langle t_s, t_p, t_o \rangle$, where t_s, t_p, and t_o are terms; thus, each triple is an atom. Moreover, $\mathsf{vars}(A)$ is the set of variables occurring in A, and $\mathsf{term}_\pi(A) = t_\pi$ for $\pi \in \Pi$. A *conjunctive query* (CQ) is an expression of the form $Q(\vec{x}) = A_1 \wedge \cdots \wedge A_n$, where $\vec{x}$ are the *answer variables* and each A_i is an atom. If $\bigcup_{A_i \in Q} \mathsf{vars}(A_i) = \vec{x}$ then Q is a *full conjunctive query*, otherwise it has *variable projection*. This definition captures *basic graph patterns* with projection in SPARQL; e.g., $Q(x) = \langle x, \mathit{rdf{:}type}, \mathit{:Person} \rangle \wedge \langle x, \mathit{:hasSister}, y \rangle$ captures the SPARQL query given above. We define a *star query* as a query consisting of only subject-subject joins.

We next define answers to a CQ $Q(\vec{x})$ on an RDF graph G. An *assignment* σ is a mapping of variables to resources. For α a term or an atom, $\alpha\sigma$ is the result of replacing each occurrence of a variable $x \in \mathsf{dom}(\sigma)$ in α with $\sigma(x)$. Moreover, σ is an *answer* to $Q(\vec{x})$ on G if an assignment ν exists such that $\sigma = \nu|_{\vec{x}}$, $\mathsf{dom}(\nu) = \mathsf{vars}(A_1) \cup \cdots \cup \mathsf{vars}(A_n)$, and $A_i\nu \in G$ for $1 \leq i \leq n$. SPARQL uses the *bag* semantics, so $\mathsf{ans}(Q, G)$ is the multiset that contains each answer σ to Q on G with multiplicity equal to the number of such ν. Finally, $|Q|_G$ is the *cardinality* of Q on G defined as the sum of the multiplicities of all answers to Q on G.

2.3 Computational Problems

Finally, we formalise the computational problems we consider in this thesis. A *cluster* C is a finite set of *servers*, connected by a network with a message passing facility, such that any server in C can send a message to any other server in C.

2.3.1 Data Partitioning

A *partition* of an RDF graph G on a cluster C is a function $\mathbf{G}$ that assigns to each server $k \in C$ an RDF graph $\mathbf{G}_k$, called a *partition element*, such that $G = \bigcup_{k \in C} \mathbf{G}_k$. Partition $\mathbf{G}$ is *strict* if each triple in G is assigned to exactly one partition element so there is no duplication, that is, $\mathbf{G}_k \cap \mathbf{G}_{k'} = \emptyset$ for all $k, k' \in C$ with $k \neq k'$. A *(data) partitioning strategy* takes an RDF graph G and a set of servers C, and produces a partition $\mathbf{G}$ of G on C.

2.3.2 Distributed Query Answering

Given a conjunctive query Q, a *distributed query answering algorithm* computes $\mathsf{ans}(Q, G)$ on a cluster C where each server $k \in C$ stores $\mathbf{G}_k$. An answer σ to Q on G is *local* if $k \in C$ exists such that σ is an answer to Q on $\mathbf{G}_k$.

2.3.3 Query Planning

Given a conjunctive query Q, and a partition $\mathbf{G}$, a *query planning algorithm* computes a reordering of the atoms, $\vec{A} = A_1, \ldots, A_n$, representing a left-deep query plan. This is the only type of query plan that we consider in this thesis.

3

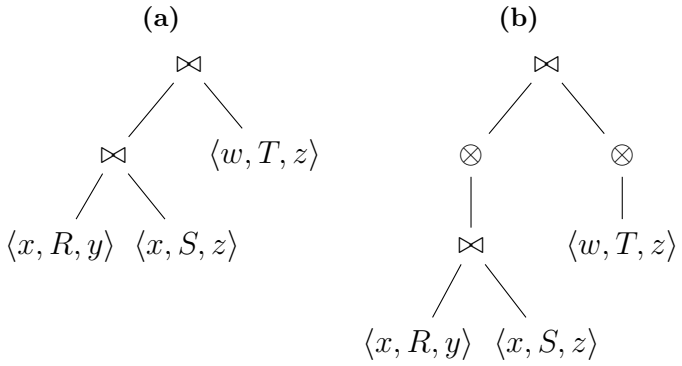
Data Exchange in Distributed Systems

One of the most important challenges that face distributed systems compared to centralised systems is data exchange. In a shared-nothing cluster, the data is partitioned across the servers, meaning that a computation could require data from several servers. It is therefore important to understand the challenges of data exchange, how existing solutions tackle these challenges, as well as the drawbacks of the those solutions. In this chapter, we thoroughly analyse the data exchange operator, which is used in most existing systems to exchange data, as well as the strategies to optimise its use. We will see that although the data exchange operator offers convenient encapsulation of communication issues, its performance is heavily dependent on the guarantees made on data placement over the cluster. Moreover, we show that the strategies employed by existing systems to combat this often incur significant storage overheads, which can affect scalability. Throughout the chapter we identify open problems and challenges that are still prevalent.

3.1 Data Exchange Operator

The *data exchange operator* was introduced in the Volcano system [25] to facilitate intra-operator parallelism on partitioned datasets. It encapsulates all communication

Figure 3.1: Example of data exchange operator in a query plan



issues between processes, allowing the reuse of existing single-process operators. When used in a query pipeline, the data exchange operator receives data from the child operator and sends it to the parent operator in the correct process(es). This is done in such a way that each operator receives all the relevant data needed for query evaluation, as illustrated in Example 1. This operator model of communication can be used for both parallel execution on a single server and distributed execution on many servers.

Example 1. *Consider the query* $Q(w, x, y, z) = \langle x, R, y\rangle \wedge \langle x, S, z\rangle \wedge \langle w, T, z\rangle$ *over an RDF graph which is partitioned across a cluster of servers in such a way that all triples with the same subject occur in the same partition element. A possible query plan for a centralised system is shown in Figure 3.1a. As the intermediate answers and triples for a given binding of variable* z *might not be on the same server, data exchange operators have to be inserted in the query plan before the join on* z*, as in Figure 3.1b, where* $\otimes$ *is the data exchange operator.*

This operator approach to parallelisation over partitioned data has seen widespread use in distributed databases and frameworks. In particular, it can be argued that the majority of modern distributed RDF databases use a variant of the data exchange operator to handle communication between servers. For this reason, it is important to understand exactly the problem data exchange operators solve, as well as how

Figure 3.2: Example partitioned RDF graph

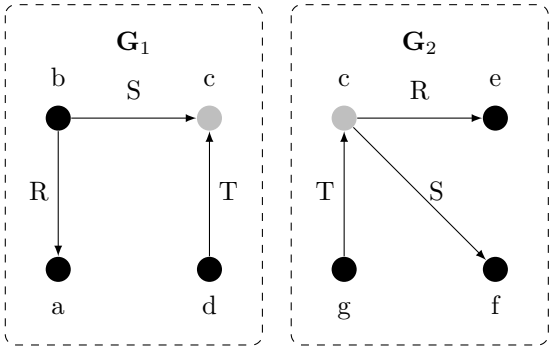
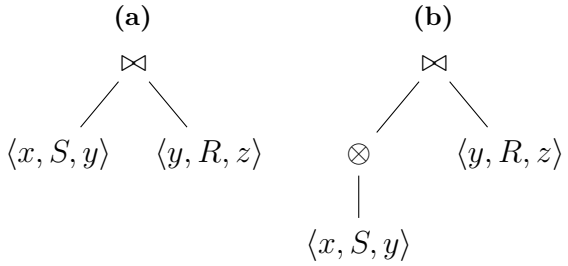


Figure 3.3: Example query plans



they do it. We do this in a concrete way through examples. Firstly, Example 2 illustrates the need for communication between servers during evaluation.

Example 2. *Let G be the RDF graph from Figure 3.2 partitioned to elements $\mathbf{G}_1$ and $\mathbf{G}_2$ by subject hashing, that is, each triple $\langle t_s, t_p, t_o\rangle$ is assigned to partition element $(h(t_s) \bmod 2) + 1$ for a suitable hash function h. Resource c is shown in grey because it occurs in both partition elements. Consider the query $Q(x, y, z) = \langle x, S, y\rangle \wedge \langle y, R, z\rangle$. Answer $\sigma = \{x \mapsto b, y \mapsto c, z \mapsto e\}$ spans partition elements as it is a result of matching triples $\langle b, S, c\rangle$ from $\mathbf{G}_1$ and $\langle c, R, e\rangle$ from $\mathbf{G}_2$. Hence, servers 1 and 2 must communicate in some way to compute σ.*

Answers such as σ in Example 2 whose data span multiple servers are referred to as *non-local answers*, which we distinguish from *local answers*, defined below.

Definition 1. *Given an RDF graph G, a partition $\mathbf{G}$ of G, and a query Q, an answer $\sigma \in \mathsf{ans}(Q, G)$ is* local *if there exists k such that $\sigma \in \mathsf{ans}(Q, \mathbf{G}_k)$, otherwise σ is* non-local.

Evaluating a query over each partition element independently will only compute local answers and miss all non-local answers. To ensure all non-local answers can be computed, data exchange operators are inserted between two operators to move data between processes. Example 3 illustrates this process.

Example 3. *Consider the RDF graph from Figure 3.2 and the query* $Q(x, y, z) = \langle x, S, y\rangle \wedge \langle y, R, z\rangle$. *Figure 3.3a shows the execution plan for* Q. *We have seen in Example 2 that this misses non-local answers, such as* σ. *There are many ways to transfer data to correctly compute the join; Figure 3.3b shows one such way. As the data is partitioned by subject hashing, given a triple from the left-hand branch such as* $\langle b, S, c\rangle$, *we know that any triples that can join to this on the right-hand branch have been hashed to partition* $(h(c) \bmod 2) + 1$. *Hence, a data exchange operator is placed between the selection* $\langle x, S, y\rangle$ *and the join on* y. *This data exchange operator then (i) sends each variable assignment from its input to server* $(h(\sigma(y)) \bmod 2) + 1$, *and (ii) receives variable assignments sent from other servers and forwards them to the parent join operator. Thus, the rest of the query plan is completely isolated from any data exchange issues.*

The data exchange operator thus solves the challenge of computing non-local answers. It also has the benefit of completely encapsulating data transfer, as other operators used in evaluation behave in the same way as if there were only a single process; they simply receive the necessary data from child operators and send their data to their parent operator as usual. However, the decision where to place data exchange operators is made *statically*, i.e. at query compile time, which can introduce inefficiencies through redundantly moving data for local answers. Data for computing local answers need not be transferred, as each local answer can be answered wholly on the server it is stored. This is illustrated in Example 4.

Example 4. *Consider the RDF graph from Figure 3.2 and the query* $Q(x, y, z, w) = \langle x, R, y\rangle \wedge \langle x, S, z\rangle \wedge \langle w, T, z\rangle$. *The join between the first two atoms can be evaluated*

locally due to subject hashing. In contrast, join variable z occurs in Q only in the object position so, given a value for z, subject hashing does not tell us where to find the relevant triples. Consequently, we need a query plan from Figure 3.1b with two data exchange operators that hash their inputs based on the value of z, which allows us to compute answers $\sigma_1 = \{x \mapsto b, y \mapsto a, z \mapsto c, w \mapsto d\}$ and $\sigma_2 = \{x \mapsto b, y \mapsto a, z \mapsto c, w \mapsto g\}$. While σ_2 is a non-local answer, σ_1 is a local answer with all necessary triples stored on server 1. However, in computing σ_1, the data exchange operators hash both sides of the join to $(h(c) \bmod 2) + 1 = 2$. This results in all the data necessary for computing σ_1 to be redundantly transferred from server 1 to server 2.

This example shows the very unintuitive side-effect of the data exchange operator, something that has not been previously mentioned in the literature. It would be much more intuitive and efficient if all local answers were computed on the server on which their respective data is stored.

3.2 Data Partitioning to Reduce Data Exchange

Data exchange is a costly operation and it can often be the bottleneck of a distributed system. Knowledge of how the data is partitioned can be used to reduce the number of data exchange operators that are used, and thus also reduce the amount of network communication. In Example 3, a data exchange operator was only placed on the left-hand side of the join, relying on the fact the data was partitioned with subject hashing. If there were no guarantee on data placement from the subject hashing, a data exchange operator would also have to be placed on the right-hand side of the join, potentially increasing the amount of data sent across the network. This observation sparked considerable research into how to partition and replicate the data in order to maximise the amount of useful guarantees on data placement. The following sections look at some of the existing partitioning strategies. We will

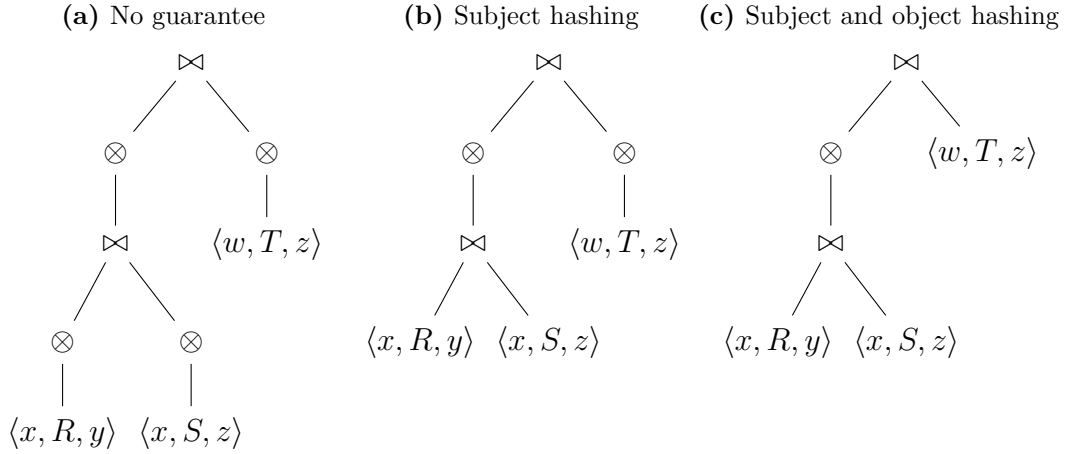
see that each strategy is aimed at a particular class of query, and thus other classes of queries receive no guarantee, resulting again in a reliance on data exchange operators.

3.2.1 Subject and Object Hashing

Subject hashing, as initially studied in the YARS2 system [31], is a very common partitioning strategy. As described earlier, it is a simple approach: for a given RDF graph G and a desired partition of size m, a hash function $h : \mathsf{voc}(G) \to \mathbb{N}$ is used to assign each triple $\langle s, p, o \rangle$ to partition element $(h(s) \, mod \, m) + 1$. Depending on the quality of the hash function used, this typically produces very balanced partitions. Furthermore, as all triples that share the same subject are placed in the same partition element, this strategy guarantees all subject-subject joins can be evaluated locally. This can be particularly useful, as studies on real world SPARQL queries have shown subject-subject joins to be very common query patterns [20], which is why subject hashing is used over object or predicate hashing.

This idea can be extended by hashing triples on both subject and object, thus replicating each triple, as used in TriAD [28]. This produces a strong guarantee because a data exchange operator is no longer needed for branches of a join that only contains a single atom. This is illustrated in Figure 3.4: when there is no guarantee on data placement, there is a data exchange operator on each branch of every join, resulting in four data exchange operators. Subject hashing reduces this number to two because the subject-subject join on x can be answered locally, only needing data exchange operators for the join on z. Finally, subject and object hashing only requires one data exchange operator, as each intermediate result from the join on x can simply be sent to the server by hashing on the value of z. Thus, this approach sacrifices storage overhead, in terms of doubling the data size, in return for reduced data exchange.

Figure 3.4: Example of the effect of different hashing strategies on query plans



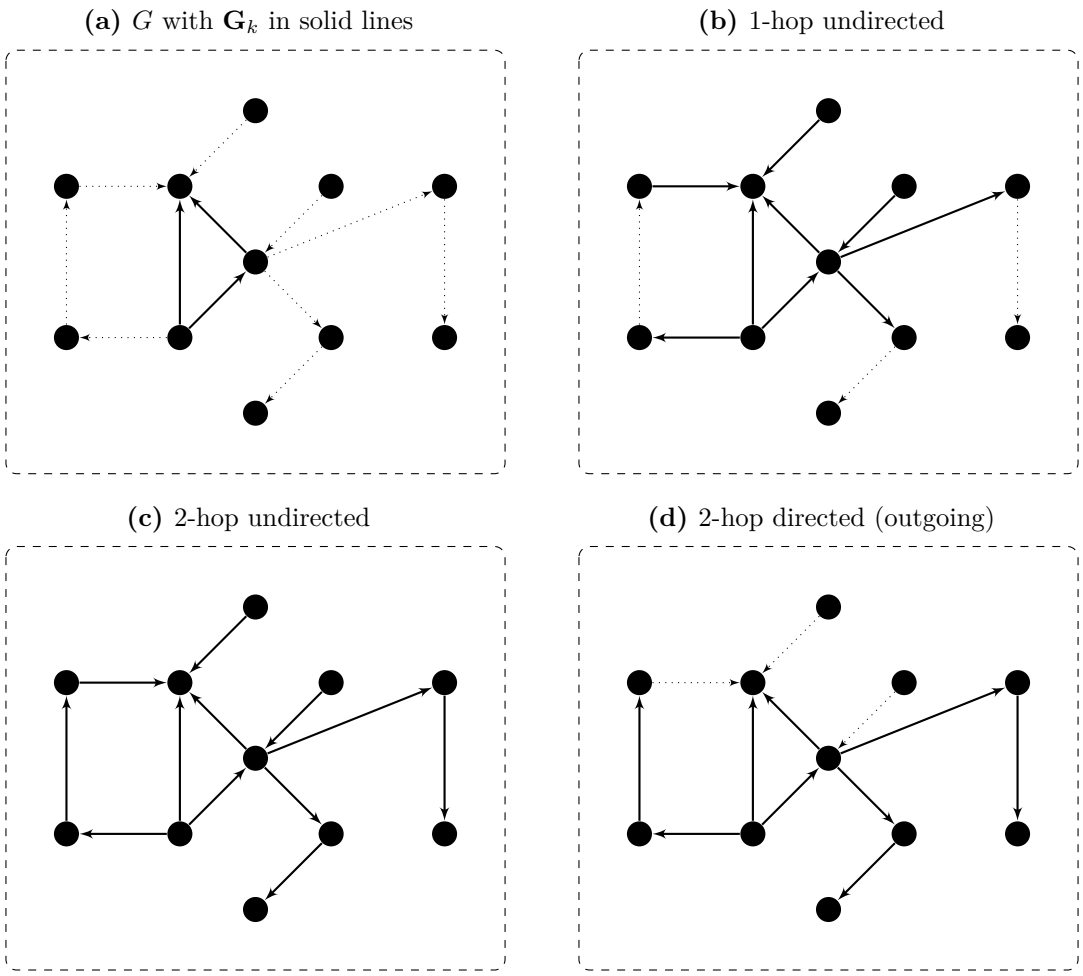
3.2.2 n-hop Guarantees

Increasing the level of useful guarantees on data placement can be achieved through n-hop guarantees, which is a data replication strategy first introduced by Huang et al. [32], and later used in systems such as SHAPE [43]. The aim is to guarantee all queries up to a certain size can be answered locally by, for each resource, replicating all data that is at most n 'hops' from the resource. Then, given a query, if there exists a term in the query that is at most n "hops" from all other terms in the query, that query can be answered completely locally. This can reduce the need for data exchange operators, creating a trade-off between storage overhead and network communication overhead. Data replication can also increase the I/O overhead during table scans, as there is more data.

Definition 2. *Given an RDF graph G and its partition $\mathbf{G}$, the* undirected n-hop guarantee *for a partition element $\mathbf{G}_i$ is defined recursively as subgraph of G, $H_i^n = H_i^{n-1} \cup \{\langle s,p,o\rangle \mid \langle s,p,o\rangle \in G, s \in \mathsf{voc}(H_i^{n-1}) \vee o \in \mathsf{voc}(H_i^{n-1})\}$, where $H_i^0 = \mathbf{G}_i$.*

The *directed n-hop guarantee* is defined in a similar way, depending on whether each resource is expanded through out-going or in-coming edges. This provides

Figure 3.5: Example of n-hop duplication



slightly more flexiblity in the storage and communication overhead trade-off, as it produces less replication, but weaker guarantees. We illustrate the different definitions in the following example.

Example 5. *Let Figure 3.5a be the graphical representation of an RDF graph G, without labels for clarity, and let the solid edges be the triples for some partition element $\mathbf{G}_k$. Then, the solid edges in Figure 3.5b are $\mathbf{G}_k$ extended with undirected 1-hop guarantee. Similarly, Figure 3.5c is undirected 2-hop guarantee and Figure 3.5d is outgoing directed 2-hop guarantee.*

Given an n-hop guarantee, a query can be answered completely locally if is sufficiently small. Defining the distance between two terms in a query as the

maximum number of (un)directed edges between the two terms in the query graph, or infinity if no path exists, a query is sufficiently small if there exists a term in the query such that its distance to every term in the query is not greater than n. In order to answer queries that are not sufficiently small, the query can be broken down into subqueries which are sufficiently small. Each subquery is then answered fully locally, then the resulting answers are joined together, such as through a MapReduce job as in [32].

The drawback of n-hop guarantees is that due to the fact RDF graphs typically have a small diameter, even small values of n create considerable levels of replication. It was shown that even an undirected 2-hop guarantee can lead to replication up to a factor of 4.54 [32]. When the goal of distributed databases is to handle large amounts of data, this level of replication can be unacceptable. Additionally, replication can lead to duplicate answers, resulting in extra information about which partition owns each resource being stored, as well as further complications in using this information during query evaluation.

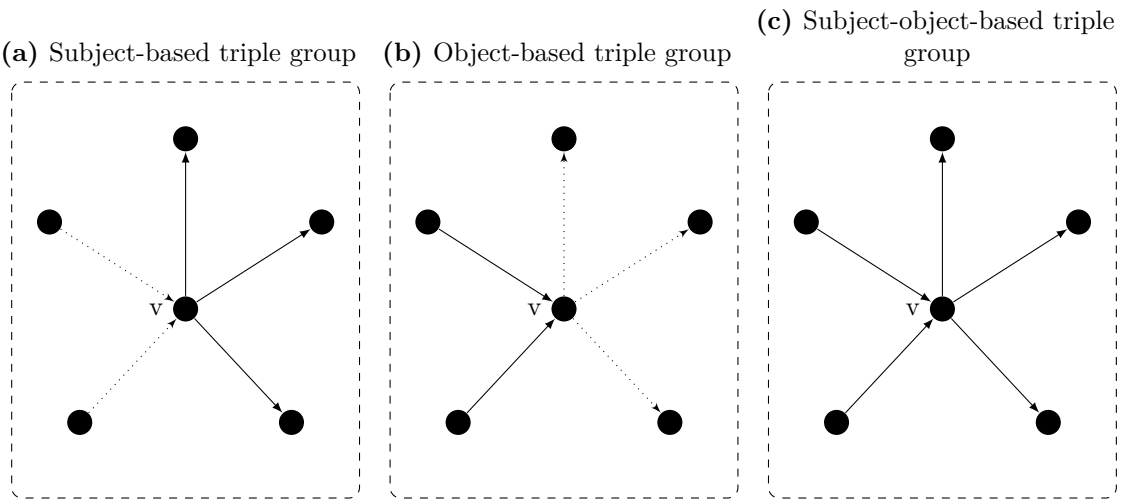
3.2.3 Triple Grouping

Triple grouping is a partitioning and replication strategy introduced in the SHAPE system [43], which takes a similar approach to n-hop guarantees. It consists of creating a *triple group* for each resource in a graph, which provides guarantees for query evaluation, and distributes those groups among partitions typically in a way that reduces the amount of data replication. The resource used to build each triple group is called the *anchor* resource. There are three types of triple group, shown in Definition 3.

Definition 3. *Let G be an RDF graph and let some $v \in \mathsf{voc}(G)$ be the anchor vertex, then;*

- *A* subject-based *triple group for v is the set $\{\langle v, p, o\rangle \mid \langle v, p, o\rangle \in G\}$.*

Figure 3.6: Example of different triple groups



- *An* object-based *triple group for v is the set* $\{\langle s, p, v\rangle \mid \langle s, p, v\rangle \in G\}$.
- *A* subject-object-based *triple group for v is the set* $\{\langle s, p, o\rangle \mid \langle s, p, o\rangle \in G, s = v \vee o = v\}$.

Example 6. *Given an RDF graph G, let v be an anchor vector for the triples represented in Figure 3.6. Then, the subject-based triple group for v is shown in Figure 3.6a as edges with a solid line. Similarly, Figure 3.6b shows the object-based triple group and Figure 3.6c the subject-object based triple group.*

These triple groups can then be used to build their corresponding partitions. The simplest method is to assign each triple group by applying a hash function to the anchor resource of each group. While this is easy to implement and typically produces balances partitions, it can also lead to significant replication. SHAPE [43] implement a URI hierarchy based optimisation to attempt to group anchor resources that are likely to have well connected triple groups, to improve query evaluation. This is done by analysing the URIs of resources. In the case of URLs, this is the domain name and subsequent paths within the URL. The assumption is then that URIs that are similar are likely to have well connected triple groups. The

authors showed that this can be particularly effective in well structured datasets, such as LUBM [27], but less effective in real world datasets.

Evaluating queries in SHAPE is again done through query decomposition. Each query that cannot be answered completely locally according to the partitioning guarantees of the triple groups is decomposed into subqueries that can. These subqueries are then evaluated independently on each server, with the intermediate resulting being joined through a MapReduce job.

3.2.4 Rooted Subgraphs

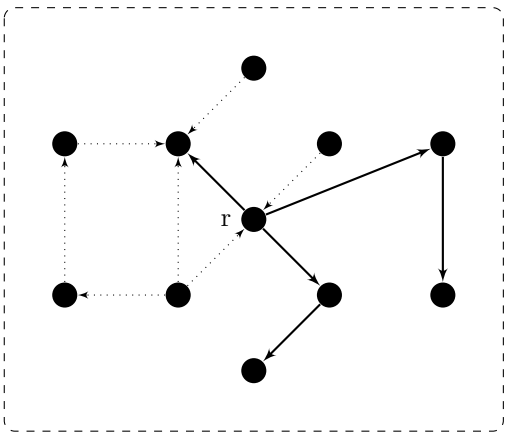
The disadvantage of n-hop guarantees and triple groups is they do not provide much benefit for complex query types. This is because triple groups only extend from the anchor by one edge, and n-hop guarantees typically can only use small values of n due to the large amount of replication. A solution to this problem is to partition using *rooted subgraphs*, introduced in SemStore [71]. A rooted subgraph for a given *root* resource (similar to an anchor resource in triple groups), is a subgraph containing all outgoing directed paths from the root.

Definition 4. *Given an RDF graph G and a root resource $r \in \mathsf{voc}(G)$, let V_r be the set all resources $v \in \mathsf{voc}(G)$ such that there is a directed path from r to v in G. The rooted subgraph of r in G is the subgraph $rsg(r)$ such that $rsg(r) = \{\langle s, p, o\rangle \mid \langle s, p, o\rangle \in G \wedge s, o \in V_r\}$.*

Example 7. *Consider the graph G and the root resource r shown in Figure 3.7. The rooted subgraph $rsg(r)$ is the subgraph of G consisting of triples shown by the solid edges.*

This creates two challenges for partitioning with rooted subgraphs. The first is selecting the set of roots such that the union of the subgraphs covers the whole graph, the second is selecting the set of roots and grouping the resulting subgraphs that maximises data locality and minimises data replication. When such a partitioning

Figure 3.7: Example of a rooted subgraph



has been chosen, query evaluation is done as before; each query is decomposed into subqueries which are guaranteed to be answered locally, with the intermediate results being joined through a MapReduce job.

3.3 MapReduce

MapReduce is a programming model, and implementation, that enables parallel and distributed processing of large datasets [14]. It has a simple interface with user-defined *map* and *reduce* primitives, similar to those used in many functional languages. The abstraction of all the complexities of distribution and network communication led to many distributed RDF databases using Hadoop [69], the open-source implementation of MapReduce. This includes systems such as that introduced by Huang et al. [32], SHAPE [43], H2RDF+ [49], HadoopRDF [33, 34], and SemStore [71]. The mechanism in which data is exchanged during processing can be seen as a variant of the data exchange operator, meaning it has the same drawbacks. Moreover, it has considerable I/O overhead which affects performance and its ability to efficiently answer ad-hoc queries.

While implementations of MapReduce can vary, the general flow of execution is largely the same. Each MapReduce job takes a list of key/value pairs as input and produces a list of keys with an associated list of values. The *map* function takes

an input key/value pair, and applies some user-defined function to it to produce an intermediate list of key/value pairs. The intermediate key/value pairs are then grouped by key and passed to the *reduce* function, where a user-defined function is applied to the list of values of a given key, emitting the result. The whole process is shown in Figure 3.8. The datatypes of these funtions are shown in (3.1) and (3.2).

$$map: \qquad (k_1, v_1) \rightarrow list(k_2, v_2) \tag{3.1}$$

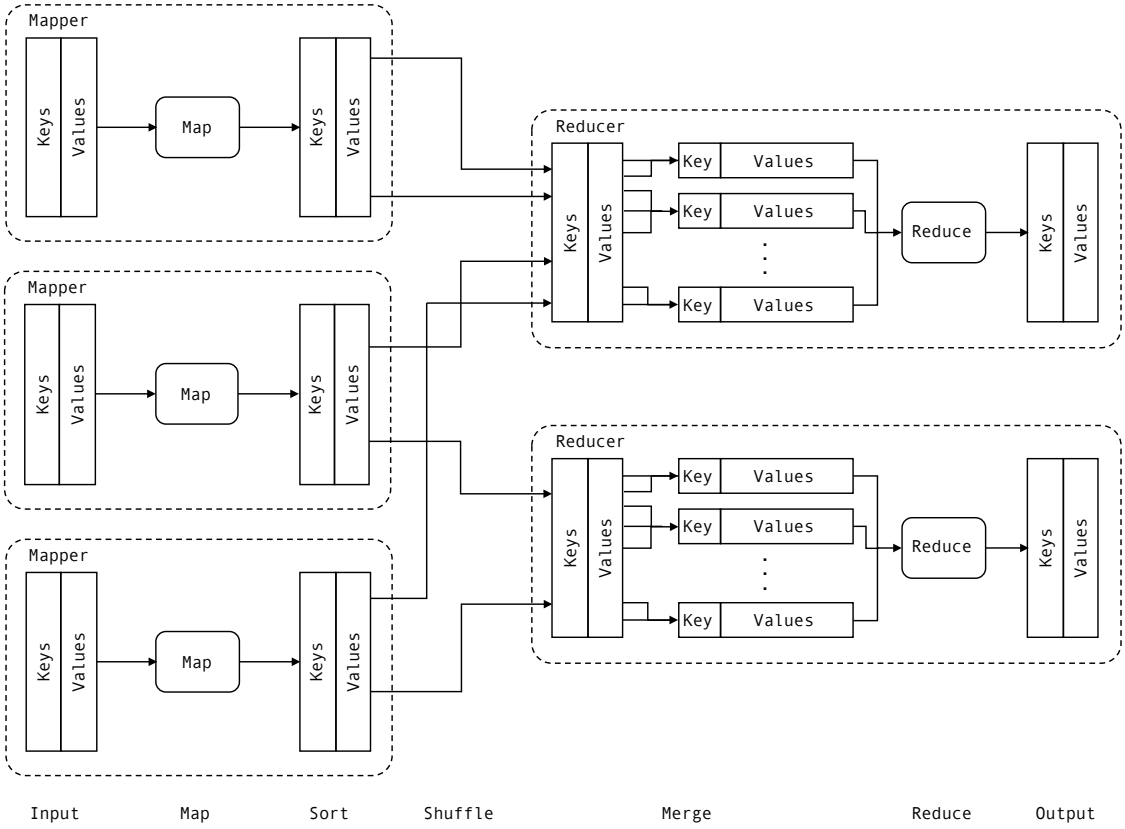
$$reduce: \qquad (k_2, list(v_2)) \rightarrow list(v_2) \tag{3.2}$$

The process between a map producing the intermediate results and the reduce consuming them is often referred to as the *shuffle* phase [69]. Each map task writes its intermediate results to the local disk, partitioning them for the subsequent set of reduce tasks. Each reducer then fetches the intermediate results from the relevant map tasks across the cluster, storing them in a mixture of memory and disk, depending on the implementation. In this sense, the shuffle phase acts in the same way as the data exchange operator: it abstracts the movement of data so that the map and reduce functions can be implemented without considering how data is moved.

While using MapReduce offers many benefits, such as ease of implementation, fault tolerance and scalability, it has many drawbacks in efficiency. There is overhead in starting and scheduling a MapReduce task, as well as the considerable I/O overhead in writing intermediate results to disk, then copying them, possibly across the network, for the reducers. While MapReduce has proven to work very well for batch processing large jobs, the associated overhead is often large for small ad-hoc queries.

Recently, an in-memory cluster computing framework Spark [73] has emerged as a successor to MapReduce. One of the goals of its design was to improve the performance of ad-hoc queries compared to MapReduce. While it provides a similar

Figure 3.8: MapReduce phases and data flow



interface to MapReduce, along with many additional features, it keeps data in memory, thus avoiding the large disk I/O costs. However, when it runs out of memory, it does spill data to disk. Spark has been used in the S2RDF system [57], where it was shown to outperform many existing MapReduce based systems such as H2RDF+ [49], Sempala [56], PigSPARQL [55], and SHARD [53].

3.4 Storage of Intermediate Answers

Distributing data over a cluster of servers to handle large volumes of data results in the total data size potentially far exceeding the resources of a single server. Furthermore, the number of answers to a conjunctive query can be exponential in the size of the data. This has implications for query evaluation whenever intermediate answers are materialised, rather than pipelined, as they could exceed

the resources of a single server and cause evaluation to fail. This is illustrated with a simple concrete example below.

Example 8. *Consider the query* $Q(x, y, z, w) = \langle x, R, y\rangle \wedge \langle x, R, z\rangle \wedge \langle w, R, z\rangle$ *evaluated over an RDF graph* $G = \{\langle a_i, R, b_j\rangle \mid i = [1..m], j = [1..n]\}$ *where each* b_i *is not necessarily distinct. Let* G *be subject hash partitioned over a cluster of size* m*, such that triple* $\langle a_i, R, b_j\rangle$ *is assigned to server* i*, then each server will hold* n *triples. Suppose the join on* x *is materialised, ready for the join on* z*, then* n^2 *intermediate answers will be materialised on each server.*

Although Example 8 is contrived, it shows that even simple queries can create huge storage requirements for materialised intermediate answers. Many existing systems materialise intermediate answers as part of their query evaluation strategy. For example, Trinity.RDF [74] materialises all intermediate answers, sending them to a single machine where a final join is performed to construct complete answers. Furthermore, every system that uses MapReduce to perform joins [32, 43, 49, 71] necessarily uses a materialisation strategy, as MapReduce writes intermediate results to disk between the map and reduce phases.

3.5 Existing Systems

We now briefly survey what we believe to be the most important existing distributed RDF databases, with extensive surveys available in the literature [18, 39]. We broadly classify systems into three categories based on their storage facility, which greatly influences how they approach both query evaluation and data partitioning. The first group consists of systems that use independent RDF stores on each server, which typically consists of a triple table and various indexes. The second group make use of key-value stores to store the RDF triples, often after some transformation. The third group use distributed file systems (DFS) to abstract away the details

Table 3.1: Comparison of distributed RDF databases

System	Storage Facility	Partitioning Scheme
Huang et al. [32]	RDF Store	Graph partitioning, n-hop guarantee
TriAD [28]	RDF Store	Subject and object hashing
PEDA [50]	RDF Store	Hashing
DREAM [29]	RDF Store	Duplicate whole graph
SHAPE [43]	RDF Store	Triple groups, n-hop guarantee
SemStore [71]	RDF Store	Rooted subgraphs
Splendid [23]	RDF Store	Federated data
Trinity.RDF [74]	Key-Value Store	Hashing
Yars2 [31]	Key-Value Store	Hashing
H2RDF+ [49]	Key-Value Store	Range partitioning
EAGRE [75]	Key-Value Store	Graph partitioning
S2RDF [57]	DFS	Extended vertical partitioning
PigSPARQL [55]	DFS	Hashing
Sempala [56]	DFS	Hashing
HadoopRDF [33, 34]	DFS	Predicate and type splitting
RAPID+ [41, 52]	DFS	Hashing
SHARD [53]	DFS	Hashing

of data access in a shared-nothing cluster, typically with Hadoop's DFS (HDFS). The systems we discuss are summarised in Table 3.1.

3.5.1 RDF Stores

This category of systems uses independent centralised RDF stores on each server, communicating by a message passing facility such as MPI [19]. Each server is assigned a subset of the data and is responsible for storing, indexing and evaluating queries over that data. One of the first systems that employed this approach was introduced by Huang et al. [32]; the system uses the graph partitioning software METIS [40] to partition the data into k sets such that the links between sets is minimised. We explore this approach in depth in Chapter 7. Furthermore, they introduced the n-hop guarantees, discussed earlier. The data is then stored in the centralised RDF database RDF-3X [46]. Their query evaluation strategy revolves around decomposing the query into subqueries that can be answered locally and

joining the resulting answers using Hadoop.

TriAD [28] uses an index-based triple store, hashing each triple by both subject and object. It also uses graph summarisation in order to implement join-ahead pruning of partition elements which contain no matching triples to the query. They use MPI as their message passing facility, to enable asynchronous communication.

PEDA [50] use the centralised RDF database gStore [76] on each of the servers, hashing the triples as their partitioning strategy. Communication is done using an implementation of MPI. They adopt a partial evaluation and assembly strategy, which we explore in greater detail in Chapter 4.

DREAM [29] uses RDF-3X as their database at each server and an implementation of MPI as their message passing facility. They employ the so-called "Quadrant-IV" approach, which does not partition the graph, but instead keeps a copy of the whole graph on disk on every server, meaning no data (apart from auxiliary coordination messages) needs to be sent over the network.

SHAPE [43] stores triples at each server using RDF-3X, partitioning the data with triple groups expanded using n-hop guarantees. They name their partitioning strategy *semantic hash partitioning* and present various optimisations to reduce the level of data duplication incurred by their partitioning strategy, such as by grouping triples through a URI-based heuristic.

SemStore [71] uses TripleBit [72] as the RDF store on each server, partitioning data using rooted subgraphs, as discussed previously. In order to reduce data duplication, they use a k-means partitioning algorithm to group rooted subgraphs that share a significant amount of triples, placing them on the same server.

Splendid [23] presents a federated approach, where each server can use an RDF database, as long as it provides a SPARQL endpoint. This is aimed at integrating disparate data sources, rather than distributing data across a cluster and hence a partitioning scheme does not apply.

3.5.2 Key-value Stores

These systems are built upon key-value stores, which use an associative array as their data model and are part of the increasingly popular NoSQL class of databases. The data consists of just key-value pairs, but richer data models can be built upon them, typically by using a tailor-made data structure for the value. Key-value stores do not normally support joins, so any such operation must be implemented at the application level.

Trinity.RDF [74] stores RDF explicitly as a graph and is built upon Trinity [61], a distributed graph database which is implemented as a distributed key-value store. The data is partitioned by hashing.

Yars2 [31] uses a key-value data model through the construction of various indexes over the data (quads rather than triples), with the key being an element of the quad. The data is hash partitioned across the cluster and query evaluation is done through index nested loop joins.

H2RDF+ [49] is built upon HBase [67], which is a distributed, key-value store. Moreover, it extends its predecessor H2RDF [48] with a more extensive indexing scheme and a different join strategy. The data is stored in the form of six indexes, according to all possible permutations of a triple's elements, as introduced by Hexastore [68]. The indexes are sorted and range partitioned. Joins are executed with merge and sort-merge joins.

EAGRE [75] uses an underlying key-value data model to build their graph-based data model. It is based on what they call RDF entities, which are pairs of resources and its one-hop neighbours. The resources then act as the key and its one-hop neighbours the values. The data is partitioned using graph partitioning software.

3.5.3 Distributed File Systems

These systems take advantage of distributed file systems, which are built to run on commodity hardware and offer a local view on distributed data by abstracting

away the details of network file access. DFS typically focus on scalability and fault-tolerance.

S2RDF [57] is based on the in-memory cluster computing framework Spark [73], which offers SQL querying. SPARQL queries are translated into SQL, according to their novel extended vertical partitioning scheme. This is an extension of vertical partitioning [1] by effectively materialising common joins.

PigSPARQL [55] is built upon MapReduce which translates SPARQL to the query language Pig Latin [47] which can be executed over Hadoop. The data is stored in triple-based files and converted at run time to the Pig Latin data model.

Sempala [56] is built on Hadoop, storing the data in tables on HDFS. Impala, which is a SQL query engine for Hadoop, is then used to query the data after the SPARQL query has been compiled to SQL.

HadoopRDF [33, 34] is built on Hadoop and stores its data in HDFS. As Hadoop's MapReduce jobs take files as input, they present a strategy for splitting the triples across files based on the predicate and type. The query joins are then executed using MapReduce.

RAPID+ [41, 52] is another Hadoop based system that relies on MapReduce to execute joins. They present techniques for increasing the amount of parallelism during query execution, reducing the number of MapReduce jobs necessary.

SHARD [53] present what they call a clause-iteration approach which assigns each clause in the query to a MapReduce job. The system is implemented in Hadoop, with the data stored in HDFS.

3.6 Conclusion

Throughout this chapter, many issues with existing systems and approaches have been identified, which are summarised below as motivation for the work in subsequent chapters.

Problem I. The static nature of data exchange operators can introduce redundant network communication. When the data for a particular answer is located on a single server, the answer will only necessarily be computed on that server if the partitioning guarantees *every* answer can be computed locally. Otherwise, data exchange operators are placed in the execution plans and redundant network communication can occur.

Problem II. Partition guarantees often trade-off storage overhead for decreased data exchange. Even weak guarantees, such as 2-hop guarantee or subject and object hashing, produce large amounts of replication that can severely limit scalability. In practice, this can mean only weak guarantees are feasible, limiting their impact on performance.

Problem III. Partition guarantees are made during setup of the system and hence before any query is issued. This means partition guarantees can only be tailored towards anticipated queries. Any unexpected, or complex query that does not fit within the guarantees receive no benefit.

Problem IV. When a query cannot be answered completely locally according to the partition guarantees, many systems decompose the query into parts which can be answered locally. This adds complexity on how best to decompose the query, as well as creates potentially very large joins when joining the intermediate answers.

Problem V. The use of MapReduce or Spark to abstract away the details of data exchange introduces considerable startup and I/O overhead which can be substantial compared to the computation cost of small, ad-hoc queries.

Problem VI. The total data size often greatly exceeds the resources of a single server in a distributed database, and moreover, the output of conjunctive queries can be exponential in the size of the data. Hence, strategies that rely on materialisation of intermediate answers are susceptible to exceeding the cluster's resources during query evaluation.

4

Wildcard Summarisation

The systems described in Chapter 3 used query evaluation strategies that were dependent on how the data was partitioned. Queries were decomposed into parts which were guaranteed to only have local answers, and variants of the data exchange operator were used to move data to join these answers to find non-local answers. We propose a new approach to distributed query answering that is independent of how the data is partitioned, based on *partial evaluation* [37]: a method of efficiently executing a program over partitioned data. Partial evaluation has been used in a wide range of applications including compilers, automatic program generation [36, 38], and more recently in answering queries over distributed XML trees [11–13] and graph simulation [17, 44].

Partial evaluation can be summarised as follows: given a program P and its partitioned input data d_1 and d_2, the goal is to compute the output $P(d_1 \cup d_2)$ by computing a program P_{d_1}, which is P partially evaluated over d_1, such that (4.1) holds true.

$$P_{d_1}(d_2) = P(d_1 \cup d_2) \tag{4.1}$$

We can apply this to our setting, where the goal is to compute the answers to Q over an RDF graph G with partition $\mathbf{G}$. Given a partition element $\mathbf{G}_k$, we want

to partially evaluate Q over $\mathbf{G}_k$ such that the output can be used to compute the answers to Q over the remaining data $G \setminus \mathbf{G}_k$.

Using partial evaluation on graphs has been previously explored in terms of reachability queries [16] and graph simulation [17, 44]. The described approaches cannot be directly applied to answering SPARQL queries over RDF graphs for two reasons: (i) RDF employs a different data model (e.g. RDF has labelled edges compared to [44]), and (ii) the semantics of the queries are different, (e.g. SPARQL employs *subgraph homomorphism*, whereas graph simulation defines a *relation* between vertices in the query graph and the graph itself). At the time the work presented in this chapter was conducted [51], no work had been done on applying partial evaluation to distributed RDF graphs. However, subsequently the system PEDA [50] has been introduced, which also applies partial evaluation to answering SPARQL queries over distributed RDF graphs.

We propose a new approach to distributed query answering based on partial evaluation that consists of two steps: (i) evaluating Q on every server in C to produce a set of partial answers to Q, and (ii) merging the partial answers to produce complete answers to Q. The key to our approach is the introduction of a *wildcard resource* $*$ which represents all resources that are external to a given server. This wildcard is used in the data to describe how partition elements are connected, as well as in partial answers to signify that they could be extended to a complete answer with data in other partition elements.

In this chapter we first introduce partial evaluation, describing partial answers in the context of RDF, how to compute them, as well as how to combine them to compute non-local answers, followed by a proof of correctness of the approach. Next, we describe a novel data partitioning scheme based on graph partitioning that aims to maximise the number of local answers to common queries in order to complement our partial evaluation approach. We evaluate our approach over

two common benchmarks and highlight its strengths as well as its limitations. Finally, we compare our approach to PEDA.

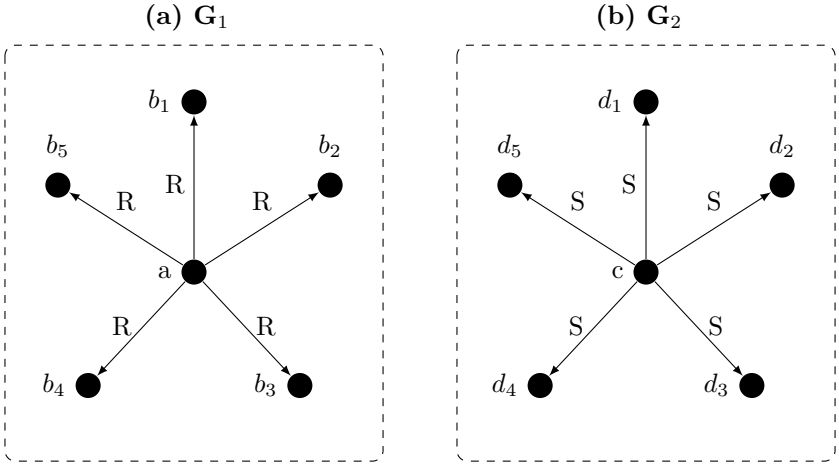
4.1 Partial Evaluation

Recall that we can distinguish between two types of answers in distributed query evaluation, those that are *local* and those that are *non-local*, where local answers are answers that can be computed independently on a single server. It is straight-forward to compute all local answers to a query over a partition; answering Q independently over each partition element and taking the union of the answers yields all local answers, $\bigcup_{k \in C} \mathsf{ans}(Q, \mathbf{G}_k)$. The challenge therefore lies in computing all of the non-local answers. We achieve this by, for each partition element $\mathbf{G}_k$, computing all partial answers to Q in $\mathbf{G}_k$, that is, all assignments that map some subset of the atoms of Q to triples in $\mathbf{G}_k$ (we formally define this notion in Section 4.1.1).

Typically, during query evaluation, the atoms of a query are matched one-by-one by extending the variable assignment from the previously matched atoms. When an atom cannot be matched, the current assignment is discarded. The naive method for computing partial answers is to simply answer Q over each partition element $\mathbf{G}_k$, without discarding the assignment when no match is found for a given atom and continuing evaluation. However, this can produce partial answers which cannot be extended to complete answers, which we refer to as *redundant*. We illustrate with the following example.

Example 9. *Let G be the RDF graph consisting of the triples $\langle a, R, b_i \rangle$ and $\langle c, S, d_i \rangle$ for $i = [1..5]$, partitioned across two servers such that all triples with predicate R are on server 1 and triples with predicate S are on server 2, as in Figure 4.1. Let $Q = \langle x, R, y \rangle \wedge \langle y, S, z \rangle$; it is clear that $\mathsf{ans}(Q, G) = \emptyset$. However, partially evaluating Q over $\mathbf{G}_1$ in the way described above produces partial answers $\sigma = \{x \to a, y \to b_i\}$*

Figure 4.1: Example of redundant answers



for $i = [1..5]$*, and over* $\mathbf{G}_2$ *produces partial answers* $\sigma = \{y \to c, z \to d_i\}$ *for* $i = [1..5]$*. Hence, all partial answers produced are redundant.*

To reduce the number of these redundant partial answers, we introduce the *wildcard resource* $*$ which, for a given partition element $\mathbf{G}_k$, represents all resources that are external to $\mathbf{G}_k$ and can be used to create an extreme summarisation of the rest of the graph $G \setminus \mathbf{G}_k$. In this way, we can encode information about how each $\mathbf{G}_k$ is connected to the rest of the graph and identify partial answers as those that match the wildcard resource. To formalise this concept, we define some notation.

Let G be an RDF graph, let the wildcard resource $*$ be a distinguished resource such that $* \notin \mathsf{voc}(G)$, and let $V \subseteq \mathsf{voc}(G)$ be a subset of the vocabulary of G. Given a resource r, let $[r]_V = r$ if $r \in V$ and $[r]_V = *$ otherwise. Additionally, given a variable x, let $[x]_V = x$. Moreover, given an atom $A = \langle t_1, t_2, t_3 \rangle$, let $[A]_V = \langle [t_1]_V, [t_2]_V, [t_3]_V \rangle$. Given a query $Q = A_1 \wedge \cdots \wedge A_n$, let $[Q]_V = [A_1]_V \wedge \cdots \wedge [A_n]_V$. Finally, given an RDF graph G, let $[G]_V$ be the RDF graph defined by $[G]_V = \{[A]_V \mid A \in G\}$. Furthermore, for μ a variable assignment, let $[\mu]_V$ be the variable assignment such that $\mathsf{dom}([\mu]_V) = \mathsf{dom}(\mu)$ and, for each variable $x \in \mathsf{dom}(\mu)$, we have $[\mu]_V(x) = \mu(x)$ if $\mu(x) \in V$, and $[\mu]_V(x) = *$ if $\mu(x) \notin V$.

To construct a partition of G, instead of partitioning triples directly, we construct n subsets of $\mathsf{voc}(G)$, $V_1, \ldots, V_n$ and use these to construct the *wildcard partition* $\mathbf{G}^* = ([G]_{V_1}, \ldots, [G]_{V_n})$. To ensure that $\mathbf{G}^*$ is a valid partition (i.e. all triples of G occur in $\mathbf{G}^*$), we must select $V_1, \ldots, V_n$ such that the following holds.

$$G \subseteq [G]_{V_1} \cup \ldots \cup [G]_{V_n} \tag{4.2}$$

Since resource $*$ is not contained in $\mathsf{voc}(G)$, we use the subset relation in condition (4.2), rather than a more intuitive equality relation. We illustrate how this construction is done in the following example.

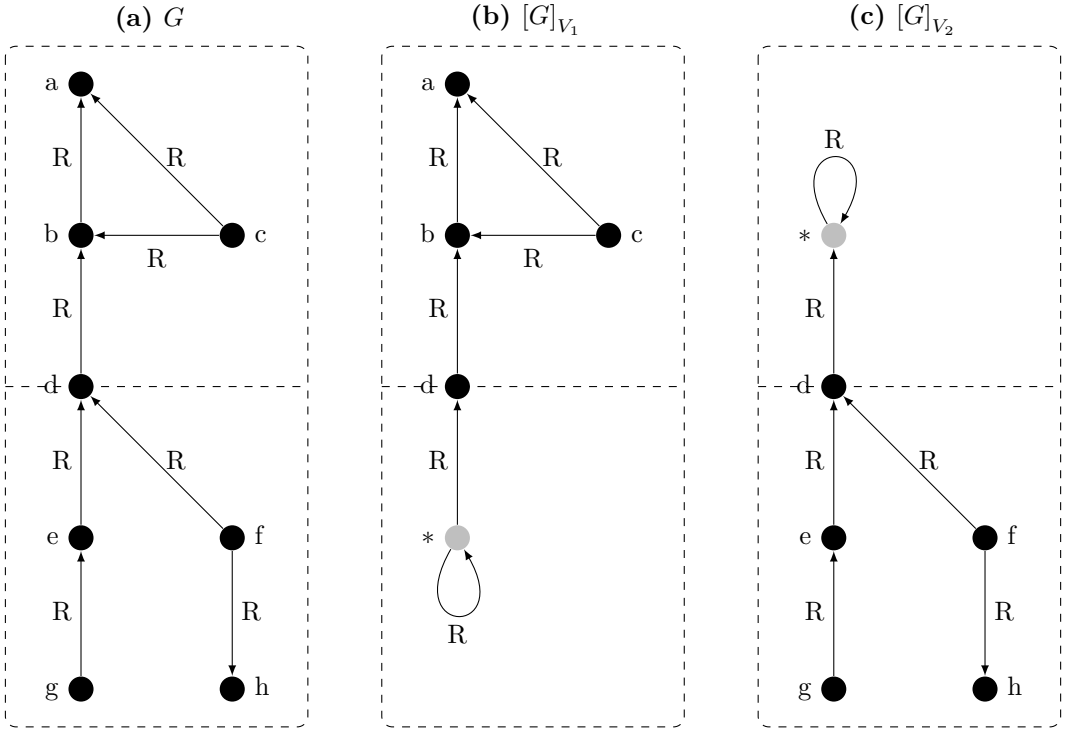
Example 10. *Let G be the graph in Figure 4.2, and let $V_1 = \{a, b, c, d\}$ and $V_2 = \{d, e, f, g, h\}$, which were chosen to satisfy* (4.2)*. Then, $[G]_{V_1}$ is the graph in 4.2b and $[G]_{V_2}$ is the graph shown in 4.2c. For $[G]_{V_1}$, the external triples (i.e. those in $G \setminus [G]_{V_1}$) are summarised as $\langle *, R, d\rangle$ and $\langle *, R, *\rangle$, and for $[G]_{V_2}$, the external triples are summarised as $\langle d, R, *\rangle$ and $\langle *, R, *\rangle$*

4.1.1 Computing Partial Answers

With the notation in place, we can define a partial answer and describe how to compute it. In order to compute partial answers, by using a wildcard resource, we completely avoid having to define any special evaluation strategy or semantics compared to standard query evaluation when dealing with atoms that do not match. That is, a standard RDF database can be used without modification to compute partial answers.

However, there might be a constant in the query that is not contained in a given partition element. That resource is represented in the data as a wildcard, hence in order for the query to correctly reflect this, the query must be rewritten by replacing all external resources with a wildcard, as in the data. That is, given a partition element $[G]_{V_k}$, Q is rewritten to $[Q]_{V_k}$. The partial answers are then

Figure 4.2: Example of wildcard summarisation



contained in the set $\mathsf{ans}([Q]_{V_k}, [G]_{V_k})$. We define partial answers in terms of *wildcard answers* below, followed by an example.

Definition 5. *Given an RDF graph G, a wildcard partition $\mathbf{G}^*$, and a query Q, a* wildcard answer *for a partition element $[G]_{V_k} \in \mathbf{G}^*$ is an assignment σ such that $\sigma \in \mathsf{ans}([Q]_{V_k}, [G]_{V_k})$ and $* \in \mathsf{rng}(\sigma) \cup \mathsf{voc}([Q]_{V_k})$. The set of wildcard answers is denoted $ans^*([Q]_{V_k}, [G]_{V_k})$.*

Example 11. *Let G and its partition be the RDF graphs in Figure 4.2 and let $Q = \langle e, R, x\rangle \wedge \langle x, R, y\rangle$. To find wildcard answers on $[G]_{V_1}$, we first transform Q to $[Q]_{V_1} = \langle *, R, x\rangle \wedge \langle x, R, y\rangle$. Answering this query over $[G]_{V_1}$ produces the wildcard answers $\sigma_1 = \{x \mapsto d, y \mapsto b\}$, $\sigma_2 = \{x \mapsto *, y \mapsto d\}$, and $\sigma_3 = \{x \mapsto *, y \mapsto *\}$.*

With the definition in place, we see that wildcard answers are partial answers that contain the wildcard resource and could possibly be extended on other servers. However, there is no guarantee that a wildcard answer can be extended on other

machines, nor is there a guarantee that if it *can* be extended, that it can be fully extended to an answer to Q. Wildcards answers that cannot be extended to answers to Q are thus redundant and it is beneficial if they can be pruned at early as possible to avoid unnecessary computation and network communication.

As mentioned earlier, local answers can be found by computing $ans(Q, [G]_{V_k})$. The following proposition relates these answers to the set of wildcard answers.

Proposition 1. *Given an RDF graph G, a wildcard partition $\mathbf{G}^*$ and a query Q, for each partition element $[G]_{V_k} \in \mathbf{G}^*$ the following holds.*

$$ans([Q]_{V_k}, [G]_V) = ans(Q, [G]_{V_k}) \cup ans^*([Q]_{V_k}, [G]_{V_k}) \tag{4.3}$$

From Proposition 1 it is immediate how to compute the wildcard answers. Computing $ans([Q]_{V_k}, [G]_{V_k})$ yields both local answers and wildcard answers. Local answers can then be output, while wildcard answers can be used to compute all non-local answers.

4.1.2 Computing Non-Local Answers

To answer a query, we first evaluate each query in each partition element independently, thus retrieving all local answers without any partial query evaluation or communication between the servers. Moreover, in this step we may also retrieve wildcard answers, each of which represents a potential match of the query across partition elements, so we merge such answers to obtain all non-local answers. In this way, we restrict communication and partial query evaluation to (possible) non-local answers, rather than all answers.

In the rest of this section, we fix an arbitrary conjunctive query Q of the form $Q = A_1 \wedge \ldots \wedge A_m$ with m atoms, an arbitrary RDF graph G, and an arbitrary wildcard partition $\mathbf{G}^* = ([G]_{V_1}, \ldots, [G]_{V_n})$ of G. To evaluate Q in $\mathbf{G}^*$, we first evaluate $[Q]_{V_i}$ in $[G]_{V_i}$ for each $1 \leq i \leq n$. Note that query Q may contain resources not contained in V_i; therefore, in each partition element $[G]_{V_i}$ we evaluate $[Q]_{V_i}$,

rather than Q. We then merge all answers obtained in the previous step, while assuming that resource $*$ matches any other resource. We formalise the merge procedure, which is an extension of the join operator, as follows.

Definition 6. *A variable assignment μ is a* merging *of assignments μ_1 and μ_2, written $\mu = \mu_1 \bowtie \mu_2$, if $\mathsf{dom}(\mu) = \mathsf{dom}(\mu_1) = \mathsf{dom}(\mu_2)$ and, for each $x \in \mathsf{dom}(\mu)$, (i) $\mu_1(x) = \mu_2(x)$ implies $\mu(x) = \mu_1(x) = \mu_2(x)$, and (ii) $\mu_1(x) \neq \mu_2(x)$ implies $\mu_1(x) = *$ and $\mu(x) = \mu_2(x)$, or $\mu_2(x) = *$ and $\mu(x) = \mu_1(x)$.*

An assignment can be an answer to Q on $\mathbf{G}^*$ only if it instantiates all atoms of Q, which we formalise as follows.

Definition 7. *Let μ be a variable assignment with $\mathsf{dom}(\mu) = \mathsf{var}(Q)$. The set of* valid atoms *of Q under μ is defined as $\mathsf{val}_\mu(Q) = \{j \mid * \notin \mathsf{voc}(\mu(A_j))\}$. Moreover, μ is* valid *for Q if $|\mathsf{val}_\mu(Q)| = m$.*

Furthermore, when evaluating $[Q]_{V_i}$ in $[G]_{V_i}$, we can ignore any variable assignment $\mu_1 \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ that is redundant according to the following Definition 8. Intuitively, μ_1 is redundant if, for each $\mu \in \mathsf{ans}(Q, G)$ that 'extends' μ_1 (i.e., such that $\mu_1 = [\mu]_{V_i}$), there exists a variable assignment μ_2 obtained by evaluating $[Q]_{V_j}$ in some partition element $[G]_{V_j}$ such that μ extends μ_2, and the set of atoms of Q fully instantiated by μ_2 strictly includes the set of atoms fully instantiated by μ_1. Note that this includes the case where no $\mu \in \mathsf{ans}(Q, G)$ extends μ_1. This idea is formally captured using the following definition.

Definition 8. *Consider arbitrary $1 \leq i \leq n$ and $\mu_1 \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$. Assignment μ_1 is* redundant *for Q and i if, for each assignment $\mu \in \mathsf{ans}(Q, G)$ such that $\mu_1 = [\mu]_{V_i}$, there exist $1 \leq j \leq n$ and an assignment $\mu_2 \in \mathsf{ans}([Q]_{V_j}, [G]_{V_j})$ such that $\mu_2 = [\mu]_{V_j}$ and $\mathsf{val}_{\mu_1}([Q]_{V_i}) \subsetneq \mathsf{val}_{\mu_2}([Q]_{V_j})$. If such μ_1 is neither valid for Q nor redundant for Q and i, then μ_1 is a* partial match of Q in $[G]_{V_i}$.

As a simple consequence of Definition 8, note that the number of non-redundant answers in each partition element is at most equal to the number of non-local query answers. Theorem 1 captures the essence of our query answering scheme. Intuitively, it says that each answer μ to Q on G is obtained either by evaluating $[Q]_{V_i}$ on some partition element $[G]_{V_i}$ (i.e., it is a local answer), or by merging non-valid and non-redundant assignments $\mu_1, \ldots, \mu_n$ obtained by evaluating $[Q]_{V_i}$ on $[G]_{V_i}$ that instantiate all atoms of Q.

4.2 Proof of Correctness

Theorem 1. *For a variable assignment μ, we have $\mu \in \mathsf{ans}(Q, G)$ if and only if*

1. *μ is valid for Q and $\mu \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ for some $1 \leq i \leq n$, or*
2. *variable assignments $\mu_1, \ldots, \mu_n$ exist such that*
   (a) *for each $1 \leq i \leq n$, either $\mathsf{dom}(\mu_i) = \mathsf{var}(Q)$ and $\mathsf{rng}(\mu_i) = \{*\}$, or we have $\mu_i \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ and μ_i is a partial match of Q in $[G]_{V_i}$,*
   (b) *for each $1 \leq j \leq m$, some $1 \leq k \leq n$ exists such that $j \in \mathsf{val}_{\mu_k}([Q]_{V_k})$, and*
   (c) *$\mu = \mu_1 \bowtie \ldots \bowtie \mu_n$.*

Proof. ($\Rightarrow$) Assume that $\mu \in \mathsf{ans}(Q, G)$. The claim holds trivially if μ satisfies (1), so assume that μ does not satisfy (1). For each $1 \leq i \leq n$, let $\xi_i = [\mu]_{V_i}$, and let μ_i be such that $\mathsf{dom}(\mu_i) = \mathsf{var}(Q)$ and $\mathsf{rng}(\mu_i) = \{*\}$ if ξ_i is redundant for Q and $\mu_i = \xi_i$ otherwise. We next show that each μ_i satisfies (2a)–(2c).

(2a) Consider an arbitrary $1 \leq i \leq n$. The claim holds trivially if ξ_i is redundant for Q and i, so we assume that $\mu_i = \xi_i$ is not redundant for Q and i. Since we assume that $\mu \notin \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$, assignment ξ_i is not valid for Q. For each $1 \leq j \leq m$, since $\mu(A_j) \in G$, we clearly have $[\mu(A_j)]_{V_i} \in [G]_{V_i}$; furthermore, we

have $[\mu(A_j)]_{V_i} = \mu_i([A_j]_{V_i})$, so $\mu_i([A_j]_{V_i}) \in [G]_{V_i}$ holds. Consequently, we have $\mu_i \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$, as required.

(2b) Consider an arbitrary $1 \leq j \leq m$; then, $\mu \in \mathsf{ans}(Q, G)$ clearly implies $\mu(A_j) \in G$. Since $G \subseteq \bigcup_i [G]_{V_i}$, some $1 \leq i \leq n$ exists such that $\xi_i(A_j) \in [G]_{V_i}$, so clearly $j \in \mathsf{val}_{\xi_i}([Q]_{V_i})$. Now choose $1 \leq k \leq n$ such that $\mathsf{val}_{\mu_k}([Q]_{V_k})$ is a maximal set satisfying $\mathsf{val}_{\xi_i}([Q]_{V_i}) \subseteq \mathsf{val}_{\xi_k}([Q]_{V_k})$. Since $\mathsf{val}_{\mu_k}([Q]_{V_k})$ is largest, such ξ_k is not redundant for Q and k, so $\mu_k = \xi_k$; but then, $j \in \mathsf{val}_{\mu_k}([Q]_{V_k})$, as required.

(2c) For each variable $x \in \mathsf{dom}(\mu)$, some $1 \leq i \leq n$ exists such that $\mu(x) \in V_i$; hence, it is obvious that $\mu = \xi_1 \bowtie \ldots \bowtie \xi_n$ holds. We next show that we can successively replace in this equation each ξ_i that is redundant for Q and i with μ_i. To this end, choose an arbitrary ξ_i that is redundant for Q and i, and choose an arbitrary $1 \leq j \leq n$ such that $\mathsf{val}_{\xi_i}([Q]_{V_i}) \subseteq \mathsf{val}_{\xi_j}([Q]_{V_j})$ and ξ_j is not redundant for Q and j; clearly, we have $[\xi_i]_{V_j} \bowtie \mu_j = \mu_j$. Now consider an arbitrary variable $x \in \mathsf{dom}(\xi_i)$ such that $\xi_i(x) \neq *$ and $\xi_i(x) \neq \xi_j(x)$. Since $\mathsf{val}_{\xi_i}([Q]_{V_i}) \subseteq \mathsf{val}_{\xi_j}([Q]_{V_j})$ holds, variable x occurs in Q only in atoms A_ℓ such that $* \in \mathsf{voc}(\xi_i(A_\ell))$, so $\ell \notin \mathsf{val}_{\xi_i}([Q]_{V_i})$. But then, by (2b), some $1 \leq k \leq n$ exists such that $\xi_k(x) = \xi_i(x)$ and ξ_k is not redundant for Q and k. But then, $\mu = \xi_1 \bowtie \ldots \xi_{i-1} \bowtie \mu_i \bowtie \xi_{i+1} \bowtie \xi_n$ clearly holds. We can iteratively replace in this equation each ξ_i that is redundant for Q and i with μ_i without affecting the equality, as required.

($\Leftarrow$) Assume that (1) is true for some μ; then $\mu([A_j]_{V_i}) = \mu(A_j)$ for each $1 \leq j \leq m$, so clearly $\mu \in \mathsf{ans}(Q, G)$. Assume now that (2a)–(2c) are true, and consider an arbitrary $1 \leq j \leq m$. By (2a) and (2b), some $1 \leq i \leq n$ exists such that $\mu_i([A_j]_{V_i}) \in [G]_{V_i}$ and $* \notin \mathsf{voc}(\mu_i([A_j]_{V_i}))$. But then, $\mu_i([A_j]_{V_i}) \in G$; furthermore, by (2c), $\mu_i(x) = \mu(x)$ for each variable $x \in V_j$, so $\mu(A_j) \in G$. Consequently, $\mu \in \mathsf{ans}(Q, G)$. □

Hence, the answers to a query Q over $\mathbf{G}$ can be computed as follows. First, each i-th server computes $\mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ in parallel, and it immediately returns all answers

that are valid for Q. Second, the server identifies a subset $P_i \subseteq \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ of partial matches of Q in $[G]_{V_i}$, and it also extends P_i with assignment μ such that $\mathsf{dom}(\mu) = \mathsf{var}(Q)$ and $\mathsf{rng}(\mu) = \{*\}$. Third, all servers communicate P_i to one designated server, which then computes the merging $P_1 \bowtie \ldots \bowtie P_n$ and returns each result that instantiates all atoms of Q. Our query answering scheme thus requires distributed computation only for answers spanning partition boundaries.

4.3 Data Partitioning

The cost of evaluating a query using partial evaluation is directly influenced by how the data is partitioned. It is clear to see that the cost of computing a non-local answer is higher than computing a local answer, since the cost of just computing a wildcard answer is the same as computing a local answer. One of the primary benefits of a partial evaluation strategy over the existing strategies described in Chapter 3 is its ability to compute local answers locally. Hence, we present a novel partitioning strategy that aims to maximise the number of local answers to common queries to demonstrate that the benefits of partial evaluation are useful and usable in practice.

The quality of partitions critically depends on the structure of the data and the anticipated query workload. Although application specific, we found the following assumptions to be common to a large number of applications; (i) subject–subject joins are common, (ii) queries often constrain variables to elements of classes—that is, they often contain atoms of the form $\langle ?x, \mathit{rdf{:}type}, \mathit{class}\rangle$, (iii) joins involving resources representing classes are uncommon—that is, queries rarely contain atoms $\langle ?x_1, \mathit{rdf{:}type}, ?y\rangle \wedge \langle ?x_2, \mathit{rdf{:}type}, ?y\rangle$, (iv) joins on resources that are literals are uncommon—that is, if a query contains atoms $\langle ?x_1, {:}R, ?y\rangle \wedge \langle ?x_2, {:}S, ?y\rangle$, it is unlikely that a query answer will map variable $?y$ to a literal, and (v) the number of schema triples in G is small, so all schema triples can be replicated in each partition element.

Although n-hop duplication can increase partition quality [32, 43], it is often associated with a considerable storage overhead. With this in mind, we formulate the following aims for our partitioning scheme; (i) maximise the number of local answers to star queries, (ii) achieve similar, or better, partition quality than schemes employing n-hop duplication, and (iii) minimise duplication, particularly compared to n-hop duplication.

Recall, that for our partitioning scheme, we must partition the vocabulary of the graph into n subsets $V_1, \ldots, V_n$ such $G \subseteq [G]_{V_1} \cup \ldots \cup [G]_{V_n}$. To achieve this, we first partition the vocabulary $\mathsf{voc}(G)$ into n disjoint sets $V'_1, \ldots, V'_n$, and then we extend these sets so that the condition is satisfied. This extension allows resources to be duplicated in multiple partition elements, which in turn means triples can also be duplicated. Typically, the duplicated triples are those that are on, or near, the border between partition elements. Furthermore, our approach ensures that, for each partition element $G_i = [G]_{V_i}$ and each triple $\langle s, p, o\rangle \in G$, if $\{s, p, o\} \subseteq \mathsf{voc}(G_i)$, then $\langle s, p, o\rangle \in G_i$ holds. This property is not satisfied in previously known partitioning schemes but it increases partition quality. We formalise these ideas using the following steps.

Step 1. Compute the undirected graph G' by removing from G all schema triples and triples containing class and literal resources (i.e., all triples of the form $\langle s, \mathit{rdf{:}type}, o\rangle$ and $\langle s, p, \ell\rangle$ with ℓ a literal), and by treating each remaining triple $\langle s, p, o\rangle$ as an undirected edge connecting s and o.

Step 2. Partition the nodes of G' into n disjoint sets using graph partitioning software (e.g. METIS [40]), and let $V'_1, \ldots, V'_n$ be the resulting vocabularies.

Step 3. Extend each V'_i to $V^\star_i = V'_i \cup \{r \mid r$ occurs in a schema triple or as a predicate in a triple in $G\}$.

Step 4. Extend each $V^\star_i$ to $V_i = V^\star_i \cup \{o \mid \langle s, p, o\rangle \in G$ and $s \in V^\star_i\}$.

Step 5. Calculate $[G]_{V_i}$ for each V_i, and set $\mathbf{G} = \{[G]_{V_1}, \ldots, [G]_{V_n}\}$.

In Step 1, we take into account the assumption that schema triples can be replicated in each partition element. Furthermore, in line with our assumptions on our query workload, we do not expect triples to often participate in joins on classes and literals; thus, we remove such triples in Step 1 so that the graph partitioning software does not attempt to place resources connected via class or literal resources into the same partition element.

In order to satisfy (4.2), we must ensure that, for each triple $A \in G$, some V_i exists such that $\mathsf{voc}(A) \in V_i$. Thus, we must reintroduce the triples from G that correspond to edges in G' that were 'cut' during partitioning, as well as triples removed in Step 1. Thus, in Step 3 we introduce all resources occurring in the schema (including all classes and properties) into all partition elements; note that this ensures an efficient evaluation of queries mentioned in Assumption 2. Finally, since we assume that subject–subject joins are common (cf. Assumption 1), in Step 4 we reintroduce the missing triples into the partition element that contains the triple subject.

Partition element $[G]_{V_i}$ is the *core owner* of a resource r if $r \in V_i'$. Note that, if $[G]_{V_i}$ is the core owner of a resource r, then $[G]_{V_i}$ contains all triples in which r occurs in the subject position. Hence, if Q is a star query in which variable $?x$ participates in subject–subject joins, then all answers in which $?x$ is mapped to r can be obtained by evaluating Q in $[G]_{V_i}$; in other words, all answers to star queries are local, which is in line with our Aim 1.

Example 12. *To make our scheme clear, we present an extended example. Let G be the RDF graph containing the following eight triples, shown schematically in Figure 4.3a.*

$$\begin{aligned} G = \{\ & \langle a, R, b\rangle, \langle b, R, c\rangle, \langle b, R, d\rangle, \langle d, R, f\rangle, \langle e, R, d\rangle, \\ & \langle f, R, a\rangle, \langle f, R, e\rangle, \langle b, \mathit{rdf{:}type}, s\rangle, \langle e, \mathit{rdf{:}type}, t\rangle\} \end{aligned} \tag{4.4}$$

To produce a 2-partition of G, in Step 1 we first remove all triples containing class and literal resources; in our example, we remove triples $\langle b, rdf{:}type, s\rangle$ and $\langle d, rdf{:}type, t\rangle$. In Step 2 we then apply graph partitioning software to the resulting graph to split the resources into two sets while minimising the number of cut edges; let us assume that this produces the following vocabularies:

$$V_1' = \{a, b, c\} \qquad V_2' = \{d, e, f\} \tag{4.5}$$

In Steps 3 and 4 we then extend these vocabularies so that each partition element that is a core owner of a subject also contains all triples with that subject, and so that each partition element includes all class and property resources; in our example, this produces the following vocabularies:

$$V_1 = \{a, b, c, d, s, t, R, rdf{:}type\} \qquad V_2 = \{a, d, e, f, s, t, R, rdf{:}type\} \tag{4.6}$$

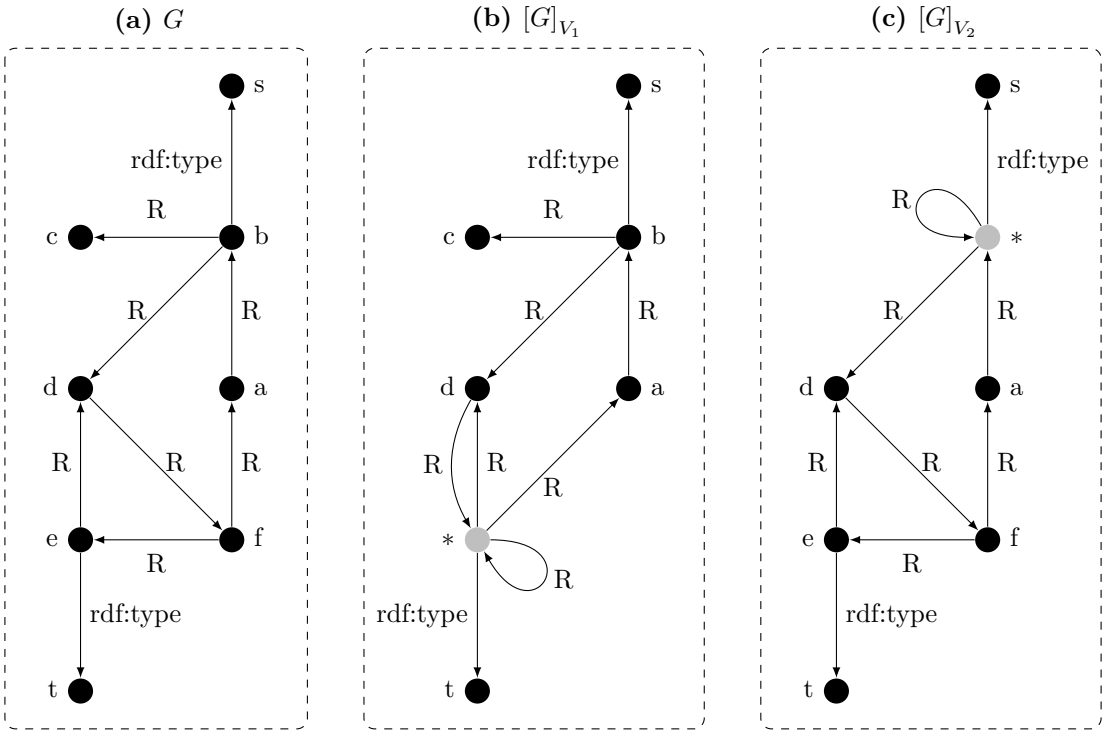
Due to this step, nodes a, d, s and t as duplicated in V_1 and V_2; we explain using node d why this is necessary. Graph $[G]_{V_1}$ must contain all triples whose subject is in V_1'; thus, since $\langle b, R, d\rangle$ is in G and b is in V_1' , we must add d to V_1. Finally, we construct $[G]_{V_1}$ and $[G]_{V_2}$ as shown in Figures 4.3b and 4.3c.

4.4 Identifying Redundant Answers

Checking whether some $\mu_1 \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$ is redundant for Q and i requires one to consider each $\mu \in \mathsf{ans}(Q, G)$, which is clearly impractical. Thus, we present an approximate redundancy check that takes advantage of the data partitioning scheme. Note that Theorem 1 holds even if some μ_i in Condition (2a) is redundant, so using an approximate check is safe from the correctness point of view.

Our optimisation is based on the fact that our data partitioning scheme ensures that answers to subject–subject joins are always local. Hence, if, for some μ, each star subquery in Q contains an atom that is not valid for μ, then μ is redundant. This is captured formally in Proposition 2; its proof is trivial.

Figure 4.3: Example of partitioning scheme



Proposition 2. *Consider an arbitrary* $1 \leq i \leq n$ *and an arbitrary variable assignment* $\mu \in \mathsf{ans}([Q]_{V_i}, [G]_{V_i})$*. Then,* μ *is redundant for* Q *and* i *if, for each term* $s \neq *$ *occurring in* $\mu([Q]_{V_i})$ *in a subject position, an atom* $A \in Q$ *exists such that* s *occurs in the subject position of* $\mu([A]_{V_i})$ *and* $* \in \mathsf{voc}(\mu([A]_{V_i}))$*.*

4.5 Experimental Evaluation

In this section we experimentally evaluate our approach by calculating (i) the percentage of local answers to test queries, (ii) the storage overhead, and (iii) the number of wildcard answers to test queries, according to Proposition 2. An important property of our approach is that it is guaranteed to compute all local answers locally, so experiment (i) tests how useful this property is in practice. We will see that using the partitioning scheme described in Section 4.3 results in a significant number of answers being local in the benchmarks used. Experiment (iii) is an effective measure of the efficiency of our approach, as the number of

wildcard answers produced influences the total amount of work necessary for query evaluation. We will see that for certain queries, a prohibitive number of wildcard answers are produced, limiting the viability of the approach.

We use the Lehigh University Benchmark (LUBM) and the SPARQL Performance Benchmark (SP2B). Each test dataset was split into a partition of size 20, and we used the queries available in the respective benchmarks. This size was chosen to make it directly comparable to related works such as [32, 43]. For a fixed dataset, increasing the partition size is likely to increase the number of non-local answers as the data becomes more fragmented; in contrast, fixing partition size while increasing the size of the dataset is likely to reduce the proportion of non-local answers. We conducted the following experiments.

We compared our approach with subject-based hash partitioning (written *Hash*) as in [31, 74], and semantic hash partitioning (written *SHAPE*) [43], which uses an optimised form of subject hashing and directed 2-hop duplication. We did not consider the graph partitioning approach by [32] because SHAPE was shown to offer superior performance. All of these partitioning approaches ensure that all answers to all star queries are local. Furthermore, Proposition 2 ensures there are no wildcard answers to star queries so we did not consider them in our tests. We use *PE* as the name of our approach.

4.5.1 Test Datasets

The Lehigh University Benchmark (LUBM) [27] is a commonly used Semantic Web benchmark. It consists of a synthetic data generator for a simple university domain ontology, and 14 test queries, nine of which are star queries. The generator is parameterised by a number of universities, for which it creates data from the university domain. We used LUBM-2000, containing approximately 267 million triples. The main drawback of LUBM is that the data for each university is highly modular: entities in each university contain many more links amongst themselves

Table 4.1: LUBM queries

Q2	SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:GraduateStudent . ?Y rdf:type ub:University . ?Z rdf:type ub:Department . ?X ub:memberOf ?Z . ?Z ub:subOrganizationOf ?Y . ?X ub:undergraduateDegreeFrom ?Y }
Q8	SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:Student . ?Y rdf:type ub:Department . ?X ub:memberOf ?Y . ?Y ub:subOrganizationOf <http://www.University0.edu> . ?X ub:emailAddress ?Z }
Q9	SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:Student . ?Y rdf:type ub:Faculty . ?Z rdf:type ub:Course . ?X ub:advisor ?Y . ?Y ub:teacherOf ?Z . ?X ub:takesCourse ?Z }
Q11	SELECT ?X WHERE { ?X rdf:type ub:ResearchGroup . ?X ub:subOrganizationOf <http://www.University0.edu> . ?DEPT ub:subOrganizationOf <http://www.University0.edu> . }
Q12	SELECT ?X ?Y WHERE { ?X ub:headOf ?Y . ?Y rdf:type ub:Department . ?X ub:worksFor ?Y . ?Y ub:subOrganizationOf <http://www.University0.edu> }
Qc	SELECT DISTINCT ?W ?X ?Y ?Z WHERE { ?X ub:worksFor ?Y . ?Y ub:subOrganizationOf ?Z . ?W ub:undergraduateDegreeFrom ?Z . ?W ub:advisor ?X }

than to entities in other universities. We used the five non-star benchmark queries, as all systems tested can answer the star queries completely locally, and a manually created circular query Qc shown in Table 4.1.

The SPARQL Performance Benchmark (SP2B) [58] is another synthetic benchmark that produces DBLP-like bibliographic data. We used an SP2B dataset with approximately 200 million triples. The benchmark provides 12 queries, of which we have used the five non-star queries for our comparison. Some of these queries contain OPTIONAL clauses, which we simply deleted because optional matches are currently not supported in our framework. These queries are shown in Table 4.2.

Both of these benchmarks use synthetic data, which has advantages and disadvantages. As the data is generated, it can be scaled to an arbitrary amount,

Table 4.2: SP2B queries

Q4	SELECT DISTINCT ?name1 ?name2 WHERE { ?article1 rdf:type bench:Article . ?article2 rdf:type bench:Article . ?article1 dc:creator ?author1 . ?author1 foaf:name ?name1 . ?article2 dc:creator ?author2 . ?author2 foaf:name ?name2 . ?article1 swrc:journal ?journal . ?article2 swrc:journal ?journal FILTER (?name1< ?name2) }
Q5	SELECT DISTINCT ?person ?name WHERE { ?article rdf:type bench:Article . ?article dc:creator ?person . ?inproc rdf:type bench:Inproceedings . ?inproc dc:creator ?person . ?person foaf:name ?name }
Q6	SELECT ?yr ?name ?document WHERE { ?class rdfs:subClassOf foaf:Document . ?document rdf:type ?class . ?document dcterms:issued ?yr . ?document dc:creator ?author . ?author foaf:name ?name }
Q7	SELECT DISTINCT ?title WHERE { ?class rdfs:subClassOf foaf:Document . ?doc rdf:type ?class . ?doc dc:title ?title . ?bag2 ?member2 ?doc . ?doc2 dcterms:references ?bag2 }

Q8	SELECT DISTINCT ?name WHERE {
	?erdoes rdf:type foaf:Person . ?erdoes foaf:name "Paul Erdoes"^^xsd:string .
	{ ?document dc:creator ?erdoes . ?document dc:creator ?author .
	?document2 dc:creator ?author . ?document2 dc:creator ?author2 .
	?author2 foaf:name ?name
	FILTER (?author!=?erdoes && ?document2!=?document &&
	?author2!=?erdoes && ?author2!=?author)
	} UNION {
	?document dc:creator ?erdoes. ?document dc:creator ?author.
	?author foaf:name ?name FILTER (?author!=?erdoes) } }

providing us control over the exact size of the data. The main disadvantage is that the data does not necessarily represent a real-world dataset in its structure, which could provide unrealistic results. This is particularly notable in LUBM, which has a small fixed structure that is repeated many times. However, LUBM is used by a significant number of RDF systems during evaluation, so it can provide a consistent point of comparison.

Table 4.3: LUBM percentage of local answers

	Q2	Q8	Q9	Q11	Q12	Qc
PE	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
SHAPE	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Hash	0.44%	4.96%	0.23%	5.80%	0.00%	0.04%

Table 4.4: SP2B percentage of local answers

	Q4	Q5	Q6	Q7	Q8
PE	95.95%	73.00%	99.90%	92.41%	91.45%
SHAPE	95.23%	9.72%	100.00%	41.97%	73.72%
Hash	0.01%	0.77%	0.25%	0.08%	0.26%

4.5.2 Partition Quality

Table 4.3 shows the percentage of local answers for each LUBM query. PE and SHAPE were able to answer all queries completely, which is in part due to the modular nature of the data; however, hashing performs poorly on all queries. Table 4.4 shows the results for SP2B. Again, hashing performs very poorly. Furthermore, both PE and SHAPE handled queries 4 and 6 well; however, PE significantly outperformed SHAPE on queries 5, 7, and 8.

One can intuitively understand these results as follows. Hashing by subject, although effective for star queries, performs very poorly for other types of query: in most cases, it provides almost no local answers. Thus, hashing is likely to be a poor data partitioning scheme for applications with diverse query loads. SHAPE considerably improves hashing, to the extent that only two benchmark queries are problematic. However, by partitioning the data based on its structure, one can further improve the overall performance: our approach is weakest on query Q5 from SP2B, but it still provides a high percentage of local answers.

Table 4.5: Storage overhead

	PE	SHAPE	Hash
LUBM	3.60%	84.23%	0.00%
SP2B	0.60%	38.63%	0.00%

4.5.3 Storage Overhead

As we have already discussed, the percentage of local answers can be increased using n-hop duplication, but at the expense of storage overhead. For example, with 2-hop duplication, the approach by [32] can incur an overhead up to 430%.

Table 4.5 shows the overhead for all partitioning schemes and data sets we considered in our experiments. Hashing clearly incurs no overhead; moreover, although SHAPE incurs a considerable overhead, that can be acceptable for some applications. Our partitioning scheme, however, exhibits a negligible overhead. Intuitively, this is due to the fact that graph partitioning software tries to group resources that are highly connected, which leads to a small level of duplication.

4.5.4 Wildcard Answers

We evaluated each test query on each partition element, and we discarded all valid assignments and redundant assignments identified according to Proposition 2. For each query, we computed the mean, minimum, maximum, and the sum of the numbers of wildcard answers across all partition elements.

On LUBM, queries Q2, Q8, Q11 and Q12 had no wildcard answers, so they can be evaluated fully locally without the need for any distributed processing. Queries Q9 and Qc had 6 and 11, respectively, partial matches in total, so the necessary distributed processing is negligible.

On SP2B, evaluating queries Q4 and Q8 on all partition elements did not finish within an hour, producing very large numbers of wildcard answers. Since the number of wildcard answers in each partition element is bounded by the number

Table 4.6: Wildcard answers for SP2B

Query	Total Answers	Non-local Answers	Wildcard Answers			
			Mean	Min	Max	Total
Q5	2,970,234	801,958	19,128	1,847	43,755	382,564
Q6	38,111,881	38,048	819,709	117,781	1,321,781	16,394,172
Q7	184,193	13,989	2,874	642	6,528	57,471

of non-local answers and the latter is small (cf. Table 4.4), this result shows that Proposition 2 is not very efficient in identifying redundant answers for queries Q4 and Q8. Table 4.6 summarises the results for the remaining queries; in order to better understand these numbers, the table also shows the numbers of total and non-local answers. For queries Q5 and Q7, the numbers of wildcard answers are much smaller than the numbers of non-local answers, suggesting that merging the partial matches should be practically feasible. In contrast, the number of partial matches to query Q6 is orders of magnitude larger than the number of non-local answers, suggesting that merging the wildcard answers might be difficult.

To summarise, our approach produces no or few wildcard answers on many types of query, but it runs into problems with long chain queries such as SP2B query Q8.

4.6 Comparison to PEDA

The work presented in this chapter, as published in Potter et al. [51] was, to the best of our knowledge, the first application of partial evaluation to answering conjunctive queries on distributed RDF graphs. Since then, the RDF database PEDA [50] has been presented which also utilises partial evaluation. We briefly compare and contrast the two approaches for completeness.

The overall approaches are very similar, consisting of a partial evaluation stage and an assembly stage. Our approach uses the wildcard resource as a special value to represent when no match occurs locally, whereas PEDA uses `NULL`. However, the approach to computing the partial answers is different.

In order to recognise a partial answer could possibly be extended on another server, both systems store information about how the partition is connected across servers. In our case, we explicitly store this information in the data as triples involving the wildcard resource. In PEDA, they duplicate resources and edges, as well as store metadata about them. Duplicated resources from other servers are marked as being external and the edges are marked as *crossing edges*, meaning they connect resources from different partition elements. With this information, they can recognise valid partial answers as those that contain crossing edges.

These differences in approaches affect how far a standard RDF database must be modified in order to support partial evaluation. In our approach, the only modification needed is substituting resources in the query with wildcards if necessary, which can be done at the application level, without any modification to the underlying RDF database. In fact, the evaluation done in Section 4.5 used an RDF store without modification. In contrast, PEDA has to modify the query answering algorithm to match `NULL`s where appropriate, as well as storing the additional metadata described above.

Another major difference occurs in the assembly stage. Both approaches propose a centralised join-based assembly that proceed in much the same way, although with different special values, wildcard and `NULL`. This was recognised in both works as being a limitation, as this could potentially be a very large and expensive join, limiting scalability. PEDA subsequently also proposed a distributed approach to assembly. Partial answers are divided into sets such that answers within a set cannot possibly join with others also in that set. These sets are then joined together to produce complete answers. They present many optimisations to finding these sets, and show that in some cases the distributed approach can outperform the centralised approach, although the difference is often not substantial.

Both approaches predictably faced the same limitations, given their similarity. The most notable was the large number of partial answers that can be produced

by some queries. The evaluation of PEDA in [50] reports large numbers of partial answers for some queries and response times up to several orders of magnitude slower than competing systems.

4.7 Conclusion

In this section we presented a novel partial evaluation strategy for answering conjunctive queries over distributed RDF graphs, using graph summarisation through the introduction of the wildcard resource. This allowed query evaluation to be independent of the partitioning scheme, which is in contrast to earlier RDF systems, which use knowledge of how the data is partitioned in devising their evaluation strategy. Our strategy consisted of two steps; first, answering modified queries independently on each server, producing all local answers and wildcard answers, and second, merging the wildcard answers to produce all non-local answers. Furthermore, we presented a data partitioning scheme to complement the query evaluation strategy by maximising the number of local answers to a query through graph partitioning, which we showed experimentally on the LUBM and SP2B benchmarks.

While we showed that graph partitioning can be an effective partitioning strategy, it has much higher computational costs than simpler strategies such as hash partitioning. In the case where the data is fixed throughout the lifetime of the system this is acceptable one-time cost. For example, partitioning the LUBM dataset took less than 30 minutes. However, when updates to the data are allowed, the cost of graph partitioning can be more problematic. Either updates place data in a computationally low-cost manner, with the side effect of possibly degrading the quality of the partitioning, or the graph partitioning is recomputed, incurring a large cost.

Practical applicability of this approach depends critically on effective removal of redundant answers. As we showed in Section 4.5, the optimisation from Proposition 2 is effective on some, but not on all queries. The latter is often the case for long chain queries (i.e., queries of the form $\langle x_0, R_1, x_1\rangle \wedge \ldots \wedge \langle x_{n-1}, R_n, x_n\rangle$): in each partition element, the wildcard resource typically has a large fan-out and fan-in, and it also occurs in triples of the form $\langle *, R_i, *\rangle$, which can give rise to a large number of answers that are not redundant.

To conclude, while effective in some cases, the approach presented in this chapter has inherent limitations, the effects of which can be observed not only in our own implementation, but also in the PEDA system. In the following chapter we will present a revised strategy that overcomes these limitations.

5
Dynamic Data Exchange

In the previous chapter we saw that partial evaluation can overcome some of the problems faced by typical distributed RDF databases, such as query evaluation being dependent on the guarantees provided by the partitioning scheme. However, it had its own limitations, most notably the creation of a large number of partial answers for some queries. In this chapter we present a novel approach to distributed query evaluation based on *dynamic data exchange* that aims to solve these limitations while retaining the important beneficial properties. We begin in Section 5.1 with an overview of our approach to query evaluation to provide an intuitive understanding, followed by a formal description of the core concepts. This includes our novel query evaluation algorithm, lightweight termination detection algorithm as well as a discussion of our guarantees on memory use. In Section 5.2 we describe optimisations to our query answering algorithm that improve response time, reduce storage overhead, and reduce network communication. Finally, in Section 5.3 we provide a proof of correctness.

5.1 Query Answering Algorithm

In the following sections let C be a cluster, let $\mathbf{G}$ be a strict partition of an RDF graph G, and let Q be a CQ with n atoms. We now show how to compute $\mathsf{ans}(G, Q)$ using dynamic data exchange. In order to make our work more accessible, in this section we present the core concepts of our approach, and in Section 5.2 we present the full, optimised version.

5.1.1 Intuition

Our algorithm can intuitively be described as an asynchronous distributed index nested loop join. Each server evaluates Q over G using nested index loop joins: starting with an empty assignment, it recursively extends the assignment by matching the atoms of Q; each assignment matching a prefix of the atoms of Q is called a *partial answer*. By letting all servers evaluate Q in parallel over their respective partition elements, we obtain all local answers to Q without any network communication or server synchronisation. To obtain the remaining answers, when a server k attempts to extend a partial answer σ to cover atom A of Q, it must take into account that other servers in the cluster may contain facts matching A as well. To this end, server k uses so-called *occurrences mappings* to identify all servers that contain all resources in A and σ. Server k forwards σ to all such servers, which then continue matching the remaining atoms of Q. The occurrences are thus used to ensure the algorithm's completeness, as well as to avoid sending σ to servers that definitely cannot extend σ to an answer to Q, which can reduce the amount of communication in the cluster.

Data exchange is thus dynamic: it is determined by the occurrences at runtime, which allows servers to always compute all local answers locally. Moreover, messages are exchanged asynchronously, without predetermined synchronisation points in

the query plan: partial answers can be sent as they are produced, which benefits parallelisation, but, as we discuss in Section 5.1.5, it also complicates termination.

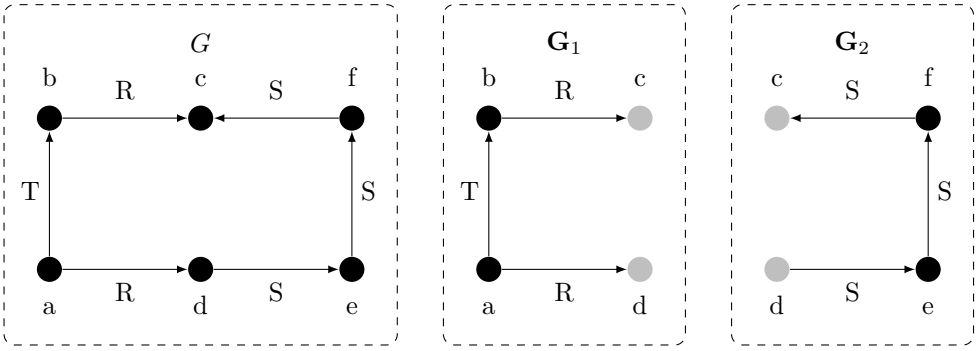
To further limit communication, resource occurrences are tracked independently for subject, predicate, and object positions. Thus, when trying to extend a partial answer σ to atom A, server k should not send σ to server k' if $A\sigma$ contains a resource at position $\pi \in \Pi$ (recall from the preliminaries that $\Pi = \{s, p, o\}$ is the set of positions of elements in a triple) that does not occur at position π in $\mathbf{G}_{k'}$. As a consequence of this optimisation, if the data is partitioned such that all triples containing the same resource in the subject are colocated, subject–subject joins can be answered without any communication. This is illustrated in the following example.

Example 13. *Consider the graph G and its partition $\mathbf{G}$ of size 2 from Figure 5.1, and the query $Q(x, y, z) = \langle x, R, y\rangle \wedge \langle z, S, y\rangle$. Evaluating the first atom of Q in $\mathbf{G}_1$ produces the partial answers $\sigma_1 = \{x \to b, y \to c\}$ and $\sigma_2 = \{x \to a, y \to d\}$, so we must next evaluate $\langle z, S, y\rangle\sigma_1 = \langle z, S, c\rangle$ and $\langle z, S, y\rangle\sigma_2 = \langle z, S, d\rangle$. By consulting the occurrences mapping, server 1 determines that, for σ_1, resources S and c occur on server 2 in the predicate and object positions respectively, so it sends a message containing σ_1 to server 2. Upon receving this message, server 2 continues evaluating the query starting from atom 2 and computes the answer $\sigma_3 = \{x \to b, y \to c, z \to f\}$. Similarly, for σ_2, server 1 determines that although server 2 contains resources S and d, resource d only occurs in the subject position, so server 2 could not extend σ_2 to an answer, so no message is sent.*

5.1.2 Setting

Before proceeding, we first discuss the technical assumptions that the servers in the cluster must satisfy. An illustration of this is shown in Figure 5.2. Any server can receive a query from a client, and that server is referred to as the coordinator for that query.

Figure 5.1: Example partitioned RDF graph

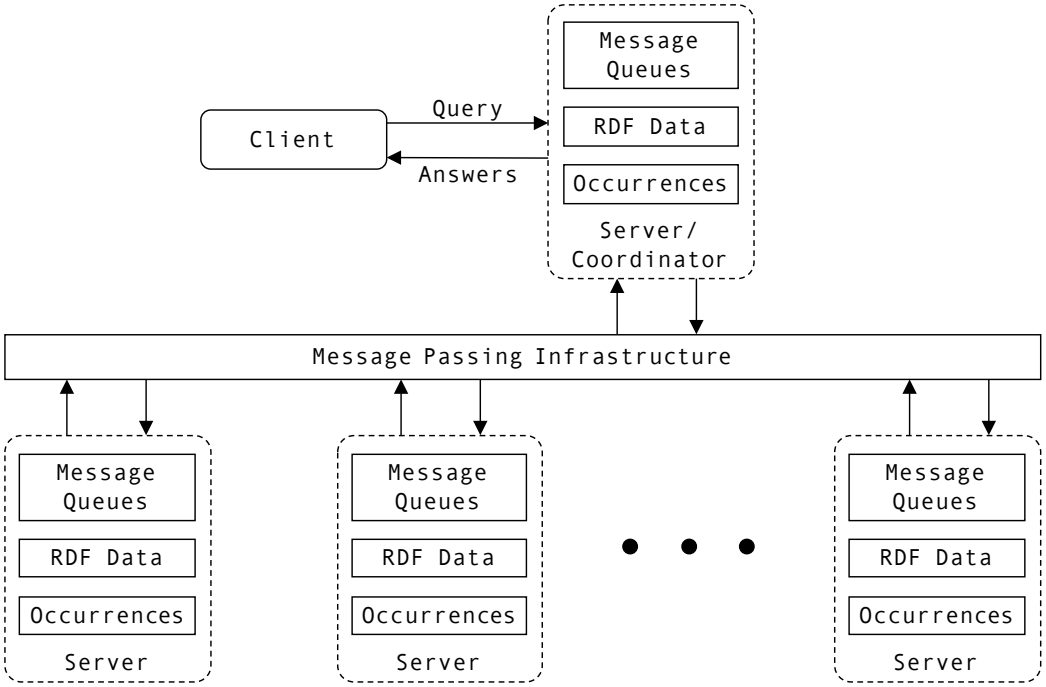


We assume that each partition element G_k is stored at server k. The query answering algorithm does not depend on any particular indexing strategy that server k may or may not use. However, for good performance, it will be beneficial that the server can efficiently match atoms. Matching atoms is done by the function $\textsc{Evaluate}(A, \mathbf{G}_k)$, which for A an atom, returns all assignments ρ such that $\mathsf{dom}(\rho) = \mathsf{vars}(A)$ and $A\rho \in \mathbf{G}_k$. To support this operation efficiently, $\mathbf{G}_k$ should be indexed. The data can be indexed in any way desirable; for example, one can use B+-trees or hash tables.

For each position $\pi \in \Pi$, server k must also store the *occurrences mapping* $\mu_\pi : \mathsf{voc}(G) \to 2^C$ that provides the *occurrences* $\mu_\pi(r) = \{k' \in C \mid r \in \mathsf{voc}_\pi(\mathbf{G}_{k'})\}$ for each resource $r \in \mathsf{voc}(G)$. The size of μ_π is determined by the number of resources and is linear in the size of G rather than a single partition element. Thus, storing μ_π on each server limits the scalability of the system. To simplify presentation of the main ideas, we first present our approach assuming each server does store μ_π and then address the issue in Section 5.2.1 by presenting a technique for only storing partial occurrences, which scales with the size of the partition element stored on the particular server.

The servers must be connected by a message passing infrastructure. To process a query with n atoms, each server must provide $n + 1$ independent message queues associated with indexes from one to $n + 1$. Each queue can be of arbitrary size, as

Figure 5.2: Architecture of a cluster



long as it can store at least one message. Then, PUTINQUEUE(ℓ, i, msg) attempts to insert message msg into queue i on server ℓ. The call returns *true* if the infrastructure can guarantee that msg will be delivered eventually; otherwise, the call returns *false* (e.g., if the destination queue is full). Finally, GETFROMQUEUE(i) extracts and returns a message from a queue with index i or higher, or it returns *null* if such queue or message does not exist.

5.1.3 Computing Query Answers

Algorithm 1 Initiating the Query at Coordinator k_c

```
1: procedure AnswerQuery(Q, x⃗)
2:     Reorder the atoms of Q as A⃗ = A_1 ∧ ··· ∧ A_n
         to obtain an efficient evaluation plan
3:     for k ∈ C do
4:         Call Start(k_c, x⃗, A⃗) on server k synchronously
5:     Send(C, PAR[1, ∅, 1, ∅⃗])
```

The client uses procedure ANSWERQUERY from Algorithm 1 to submit the query Q to any server k_c, which becomes the *coordinator* for Q. The coordinator first

Algorithm 2 Processing at Server k

6: **procedure** START($k_c, \vec{x}, \vec{A}$)
7: **for** $1 \leq i \leq n+1$ **do**
8: **for** $\ell \in C$ **do** $S_{i,\ell} := 0$
9: $P_i := 0$
10: $R_i := (i = 1 \ ? \ 1 : 0)$
11: $N_i := (i = 1 \ ? \ |C| : 0)$
12: $F_i := \mathit{false}$
13: PREPAREOPTIMISATIONS($\vec{x}, \vec{A}$)
14: Start message processing threads

15: **procedure** PROCESSMESSAGE(msg)
16: **if** $msg = \mathsf{PAR}[i, \sigma, m, \vec{\lambda}]$ **then**
17: MATCHATOM($i, \sigma, m, \vec{\lambda}$)
18: $P_i := P_i + 1$
19: CHECKTERMINATION(i)
20: **else if** $msg = \mathsf{ANS}[\sigma, m]$ **then**
21: Output answer $\langle \sigma, m \rangle$ to the client
22: $P_{n+1} := P_{n+1} + 1$
23: CHECKTERMINATION($n+1$)
24: **else if** $msg = \mathsf{FIN}[i, S]$ **then**
25: $N_i := N_i + 1$
26: $R_i := R_i + S$
27: CHECKTERMINATION(i)

28: **procedure** CHECKTERMINATION(i)
29: **if** $P_i = R_i$ **and** $N_i = |C|$ **and** SWAP(F_i, true) $= \mathit{false}$ **then**
30: **if** $i = n+1$ **then**
31: Tell client that Q has been answered and **exit**
32: **else if** $i = n$ **then**
33: SEND($\{k_c\}, \mathsf{FIN}[n+1, S_{n+1,k_c}]$)
34: **if** $k \neq k_c$ **then exit**
35: **else**
36: **for** $\ell \in C$ **do** SEND($\{\ell\}, \mathsf{FIN}[i+1, S_{i+1,\ell}]$)

37: **procedure** SEND(L, msg)
38: $i :=$ the stage index of msg
39: **for** $\ell \in L$ **do**
40: **while** PUTINTOQUEUE(ℓ, i, msg) $= \mathit{false}$ **do**
41: $msg' :=$ GETFROMQUEUE($i+1$)
42: **if** $msg' \neq \mathit{null}$ **then** PROCESSMESSAGE(msg)
43: **if** msg is a PAR or an ANS message **then** $S_{i,\ell} := S_{i,\ell} + 1$

determines an efficient left-deep join plan (line 2); we discuss planning issues in detail in Chapter 6. The coordinator then distributes the query plan to all servers synchronously (line 4), which ensures that all servers are ready before distributed processing starts. Finally, the coordinator sends to each server in the cluster the empty partial answer to kick-start processing (line 5) and then returns control to the client. The client will receive $\mathsf{ans}(Q, G)$ asynchronously (i.e. as answers are produced) in the form of pairs $\langle \sigma, m \rangle$, where σ is an assignment and m is a positive integer called the *multiplicity*. Some σ can be output more than once, but the corresponding multiplicities will add up to the multiplicity of σ in the multiset $\mathsf{ans}(Q, G)$. The basic algorithm in this section always produces $m = 1$, but this will change with optimisations from Section 5.2.

Each server $k \in C$ (including the coordinator) accepts Q for processing using the procedure Start in Algorithm 2. The procedure initialises several variables (lines 7–12), prepares data structures used in the optimisations we discuss in Section 5.2 (line 13), starts a number of message processing threads (line 14), and then terminates. All further processing at server k is driven by message processing threads, which repeatedly call GetFromQueue(1) and pass the result to ProcessMessage. There are three message types.

- $\mathsf{PAR}[i, \sigma, m, \vec{\lambda}]$ says that σ is a partial answer up to the i-th atom of Q, with integer multiplicity m and partial occurrence mappings $\vec{\lambda}$ (cf. Section 5.2).
- $\mathsf{FIN}[i, S]$ is used to detect termination. It signals that stage $i - 1$ has been processed by a server where S is the total number of partial answers for stage i sent to the receiving server (cf. Section 5.1.4).
- $\mathsf{ANS}[\sigma, m]$ says that σ is an answer to the query Q with multiplicity m.

Each message has an integer *stage*: the stage of PAR and FIN messages is i, and the stage of ANS messages is $n + 1$.

Algorithm 3 Simplified Atom Matching at Server k

```
44: procedure MatchAtom(i, σ)
45:     for ρ ∈ Evaluate(A_i σ, G_k) do
46:         σ' := σ ∪ ρ
47:         if i = n then
48:             if k = k_c then
49:                 Output answer ⟨σ'|_x⃗, 1⟩ to the client
50:             else
51:                 Send({k_c}, ANS[σ'|_x⃗, 1])
52:         else
53:             L := Occurs(A_{i+1} σ')
54:             Send(L \ {k}, PAR[i + 1, σ', 1, ∅⃗])
55:             if k ∈ L then MatchAtom(i + 1, σ')
56: function Occurs(A)
57:     L := C
58:     for π ∈ Π and t = term_π(A) do
59:         if t ∈ dom(μ_π) then L := L ∩ μ_π(t)
60:     return L
```

A partial answer message $\mathsf{PAR}[i, \sigma, m, \vec{\lambda}]$ is passed to the MatchAtom procedure, but parameters m and $\vec{\lambda}$ are only used by the optimisations described in Section 5.2. The procedure computes all extensions of the partial answer σ to atoms $A_i, \ldots, A_n$ of Q using recursive index nested loop joins. It evaluates $A_i\sigma$ in $\mathbf{G}_k$ (line 45) and, for each match ρ, it extends σ with ρ to an assignment σ' covering all atoms of Q up to A_i. The recursion base is given by $i = n$ (line 47): σ' is then an answer to Q on G, so the projection $\sigma'|_{\vec{x}}$ of σ' to the answer variables $\vec{x}$ is output to the client if server k is the coordinator (line 49); otherwise, $\sigma'|_{\vec{x}}$ is sent to the coordinator (line 51). If $i \neq n$, atom $A_{i+1}\sigma'$ must be matched recursively, and this might be done on servers other than k due to data distribution. Function Occurs identifies the set L of the relevant servers (line 53): server ℓ is included in L only if, for each position $\pi \in \Pi$ such that $t = \mathsf{term}_\pi(A_{i+1}\sigma')$ is a resource, we have $\ell \in \mu_\pi(t)$—that is, resource t occurs in partition element $\mathbf{G}_\ell$ in position π. Computation is then branched to each server in $L \setminus \{k\}$ via a $\mathsf{PAR}[i + 1, \sigma', 1, \vec{\emptyset}]$ message (line 54); moreover, matching also proceeds on server k via a recursive call if $k \in L$ (line 55).

5.1.4 Detecting Termination

Eliminating global coordination benefits parallelisation, but complicates termination: even if a server is (temporarily) idle, it could be reactivated by receiving a partial answer message. Our query answering algorithm addresses this issue by a novel asynchronous termination condition.

Server k finishes processing stage i when (i) all servers have finished processing stages up to $i-1$, and so will not send server k any more partial answers for stage i, and (ii) it has processed all received partial answers for stage i. To detect this condition, when server ℓ finishes processing a stage $i-1$, it sends to server k a $\mathsf{FIN}[i, S]$ message, where S is the total number of partial answers for stage i that server ℓ sent to server k. Server k processes this message by incrementing counter N_i and adding S to counter R_i (lines 25–26); thus, N_i counts the servers that informed server k of finishing stage $i-1$, and R_i counts the messages that server k will receive for stage i. Thus, when $N_i = |C|$ holds at server k, all other servers have processed stages up to $i-1$. Moreover, whenever server k processes a partial answer message for stage i, it increments a counter P_i (line 18 or 22). If $P_i = R_i$ also holds, then server k has finished stage i, and it sends a FIN message to the coordinator if $i = n$ (line 33) or to each server ℓ otherwise (line 36); these FIN messages include the number of sent partial answers, so these are also counted (line 43). Operation $\textsc{Swap}(F_i, \mathit{true})$ (line 29) atomically stores *true* into F_i and returns the previous value of F_i; thus, *false* is returned just once, which ensures that only one thread on server k sends termination messages. Each server sends a message to all other servers per stage, so detecting termination requires $\Theta(n|C|^2)$ messages.

5.1.5 Memory and Termination Guarantees

A nested index loop join in a centralised system requires one iterator per query atom. A query with n atoms thus needs $O(n)$ memory, which is independent of the data size and is particularly important in RAM-based RDF stores. However, our distributed

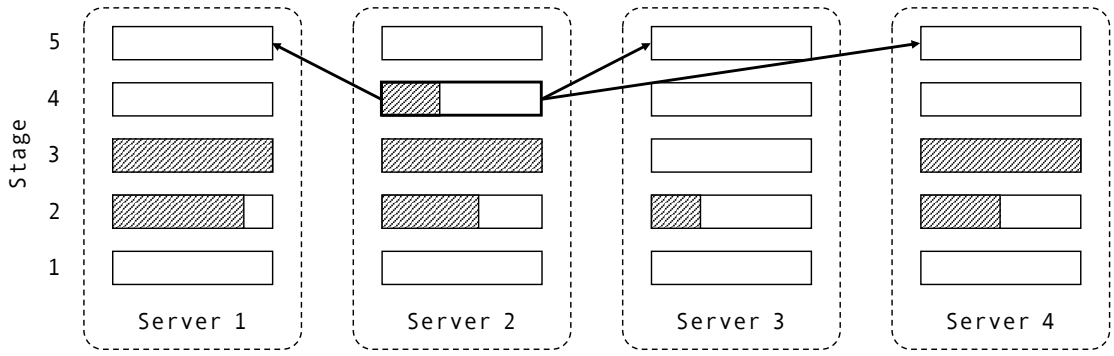
algorithm does not exhibit this property: partial answers produced in line 54 must be stored in queues of the receiving server. If G is of size m, a query with n atoms can produce m^n answers in the worst case, so the cumulative size of all messages sent to a server can easily exceed the server's capacity. Flow control techniques such as the sliding window protocol can limit the space required, but they can easily lead to deadlocks where two servers wait indefinitely for the other server's queues to free up.

We address this problem via more sophisticated flow control. As mentioned in Section 5.1.2, for a query with n atoms, each server must provide $n + 1$ distinct message queues, one per stage. Function SEND delivers each message msg to the queue determined by the stage of msg, and it will try doing so until the target queue becomes free (lines 40–42). To avoid deadlocks, after each unsuccessful delivery attempt, the function will process a message for stages $i + 1$ to n, if any exist. The recursion depth of each thread is thus $O(n)$, and each level requires $O(n)$ memory, so we need at most $O(n^2)$ memory per thread: more memory can benefit parallelism, but it is not strictly needed for the query.

We next argue that this solution avoids deadlocks, regardless of the message queue sizes. First, processing a message for stage i can produce messages only for stages $j > i$. Second, at any given point in time the cluster contains at least one highest-indexed nonempty queue across the cluster, and messages from this queue can always be processed. Thus, although individual servers in the cluster can become blocked at different points in time, at least one server in the cluster makes progress at any given point in time, which eventually ensures termination. This is illustrated in the following example.

Example 14. *Consider a set of four servers, answering a query with five stages, as illustrated in Figure 5.3. This depicts the message queues for each server and stage, with the shaded region representing how full the queue is. It is possible that server 3 has become blocked, if the messages for stage 2 only need to be sent to stage 3 (it*

Figure 5.3: Message buffers and data flow



is possible messages from stage 2 need to be sent to stages 3, 4 or 5). Regardless of servers being blocked, there always exists at least one highest-indexed nonempty queue, which is stage 4 on server 2 in this case. Processing a message for stage 4 can only produce messages for stage 5, which are necessarily empty as stage 4 has the highest-indexed nonempty queue. Hence these messages can not be blocked in the current state.

5.2 Optimisations

In Section 5.1 we described the main ideas behind dynamic data exchange. In this section, we present several optimisations that avoid unnecessary computation, and reduce storage requirements and network communication. Algorithm 4 shows the optimised version of the MatchAtom procedure, as well as the PrepareOptimisations procedure that precomputes data structures used during optimised atom matching. Thus, the optimised variant of our approach consists of Algorithms 1, 2, and 4.

5.2.1 Partial Occurrences

In Section 5.1.2 we assumed that server k stores occurrences mappings covering all resources of G. The size of G, rather than the size of $\mathbf{G}_k$, determines the memory requirements of server k, which can be detrimental to scalability.

To address this, we let server k store only the mapping $\mu_{k,\pi} = \mu_\pi|_{\mathsf{voc}(\mathbf{G}_k)}$ for each position $\pi \in \Pi$. As $\mu_{k,\pi}$ covers only the resources contained in $\mathsf{voc}(\mathbf{G}_k)$, the size of $\mu_{k,\pi}$ is determined by the size of $\mathbf{G}_k$. Note that $\mu_{k,\pi}(r)$ is defined for a resource $r \in \mathsf{voc}(\mathbf{G}_k)$ even if $r \notin \mathsf{voc}_\pi(\mathbf{G}_k)$. This, however, introduces a problem which we illustrate with the following example.

Example 15. *Let Q and a partition of size 2 be as below.*

$$Q_4(x) = \langle x, R, y\rangle \wedge \langle y, S, z\rangle \wedge \langle x, S, w\rangle$$
$$\mathbf{G}_1 = \{\langle a, R, b\rangle, \langle a, S, d\rangle\} \qquad \mathbf{G}_2 = \{\langle b, S, c\rangle\}$$

Matching the first two atoms in $\mathbf{G}_1$ and $\mathbf{G}_2$, respectively, produces a partial answer $\sigma = \{x \mapsto a, y \mapsto b, z \mapsto c\}$. Applying σ to the third atom of Q produces $\langle a, S, w\rangle$, so we need the occurrences of a and S to determine how to proceed. However, resource a does not occur in $\mathbf{G}_2$ so $\mu_{2,s}(a)$ is undefined. Thus, we can only ignore a and send σ to all servers containing S, which is likely to be inefficient.

To address this, each message $\mathsf{PAR}[i, \sigma, m, \vec{\lambda}]$ includes a vector $\vec{\lambda} = \lambda_s, \lambda_p, \lambda_o$ of *partial occurrence mappings*, where each λ_π records the occurrences of the resources appearing in $A_{i+1}\sigma, \ldots, A_n\sigma$ at position π. Note that, if server k receives such a message, it does not need the occurrences of terms appearing only in $A_i\sigma$: if server k extends σ to σ' by matching $A_i\sigma$ in $\mathbf{G}_k$, only the occurrences of resources appearing in $A_{i+1}\sigma', \ldots, A_n\sigma'$ will be used to determine where to send any subsequent messages. Moreover, $\mu_{k,\pi}$ clearly maps the occurrences of the resource appearing at position π in $A_i\sigma'$, so this information can be added to λ_π; the resulting mappings are then filtered so that only those concerning resources occurring in some $A_{i+2}\sigma', \ldots, A_n\sigma'$ are sent to other servers (line 82). In contrast, the recursive call in line 85 uses $\vec{\lambda}$ instead of $\vec{\lambda}'$; this is necessary for the correct operation of backjumping in line 70 (see Section 5.2.3).

Function $\textsc{Occurs}(A, \vec{\lambda})$ combines the occurrences $\vec{\lambda}$ sent to server k with the occurrences $\mu_{k,\pi}$ stored locally to identify servers that can possibly match A. Note

Algorithm 4 Optimised Atom Matching at Server k

61: **procedure** PREPAREOPTIMISATIONS($\vec{x}, \vec{A}$)
62: **for** $x \in \mathsf{vars}(\vec{A})$ **do**
63: $I_x := \min\{j \mid x \in \mathsf{vars}(A_j)\}$
64: $D_x := \{\pi \in \Pi \mid \exists i > I_x$ such that $x = \mathsf{term}_\pi(A_i)\}$
65: **for** $1 \le i \le n$ **do**
66: $B_i := \max[\{0\} \cup \{I_x \mid x \in \mathsf{vars}(A_i)$ and $I_x < i\}]$
67: $X_i := \vec{x} \cup \mathsf{vars}(A_{i+1}) \cup \cdots \cup \mathsf{vars}(A_n)$
68: **function** MATCHATOM($i, \sigma, m, \vec{\lambda}$)
69: $E :=$ EVALUATE($A_i\sigma, \mathbf{G}_k, X_i$)
70: **if** $E = \emptyset$ **and** $B_i \neq i - 1$ **and** OCCURS($A_i\sigma, \vec{\lambda}$) $\subseteq \{k\}$ **then return** B_i
71: **for** $\langle \rho, c \rangle \in E$ such that CANPRUNE(ρ) = *false* **do**
72: $\sigma' := (\sigma \cup \rho)|_{X_i}$
73: $m' := m \cdot c$
74: **if** $i = n$ **then**
75: **if** $k = k_c$ **then**
76: Output answer $\langle \sigma', m' \rangle$ to the client
77: **else**
78: SEND($\{k_c\}$, ANS$[\sigma', m']$)
79: **else**
80: $L :=$ OCCURS($A_{i+1}\sigma', \vec{\lambda}$)
81: **for** $\pi \in \Pi$ **do**
82: $\lambda'_\pi := (\lambda_\pi \cup \mu_{k,\pi})|_Y$ where
$Y := \{\mathsf{term}_\pi(A_j\sigma') \mid i + 1 < j \le n\}$
83: SEND($L \setminus \{k\}$, PAR$[i + 1, \sigma', m', \vec{\lambda}']$)
84: **if** $k \in L$ **then**
85: $j :=$ MATCHATOM($i + 1, \sigma', m', \vec{\lambda}$)
86: **if** $j < i$ **then return** j
87: **return** $i - 1$
88: **function** CANPRUNE(ρ)
89: **for** $x \in \mathsf{dom}(\rho)$ and $\pi \in D_x$ **do**
90: **if** $\mu_{k,\pi}(\rho(x)) = \emptyset$ **then return** *true*
91: **return** *false*
92: **function** OCCURS($A, \vec{\lambda}$)
93: $L := C$
94: **for** $\pi \in \Pi$ and $t = \mathsf{term}_\pi(A)$ **do**
95: **if** $t \in \mathsf{dom}(\lambda_\pi)$ **then** $L := L \cap \lambda_\pi(t)$
96: **else if** $t \in \mathsf{dom}(\mu_{k,\pi})$ **then** $L := L \cap \mu_{k,\pi}(t)$
97: **return** L

that resources on which neither $\lambda_\pi(r)$ nor $\mu_{k,\pi}(r)$ are defined are skipped. Thus, if atom $A_{i+1}\sigma'$ contains a resource r at position π that appears in neither λ_π nor $\mu_{k,\pi}$ (which can occur if there is a constant in the query), set L in line 80 can actually contain a server ℓ such that $\ell \notin \mu_\pi$, but this can only cause superfluous PAR messages to be sent in line 83.

5.2.2 Projecting Out Variables Eagerly

Projecting our variables eagerly can reduce both local processing and network communication, as we illustrate next.

$$\mathbf{G}_1 = \{\langle a, S, b_i\rangle \mid 1 \le i \le u\} \quad Q_5(x) = \langle x, S, y\rangle \wedge \langle x, T, z\rangle$$
$$\mathbf{G}_2 = \{\langle a, T, c_j\rangle \mid 1 \le j \le v\}$$

SPARQL uses the bag semantics, so Q_5 has just one answer $\sigma = \{x \mapsto a\}$ with multiplicity $u \cdot v$, and our unoptimised algorithm uses $u \cdot v$ steps and u messages to compute it. We can, however, evaluate $\langle x, S, y\rangle$ in $\mathbf{G}_1$ and project out y, and thus obtain just one partial answer $\sigma_1 = \{x \mapsto a\}$ with multiplicity u. We send both to $\mathbf{G}_2$, and then we evaluate $\langle a, T, z\rangle$ and project out z, and thus obtain just one match $\rho = \emptyset$ with multiplicity v. Finally, we combine σ_1 and u with ρ and v into the answer $\sigma = \sigma_1 \cup \rho$ with multiplicity $u \cdot v$. We thus need only $u + v$ steps and just one message.

Grouping assignments after variable projection can be costly and can prevent parallelisation, so we only project out variables when matching atoms. In particular, we match an atom A in a graph G using function EVALUATE(A, G, X), where set X contains the relevant variables of A. The function returns a set of pairs $\langle \rho, c\rangle$ where ρ is an assignment with $\mathsf{dom}(\rho) = X \cap \mathsf{vars}(A)$ and c is the (positive) number of distinct extensions ρ' of ρ such that $A\rho' \in G$. For example, EVALUATE$(\langle x, R, y\rangle, \mathbf{G}_1, \{x\})$ returns just one pair $\langle \{x \mapsto a\}, u\rangle$. This operation can be easily implemented in RDF stores with hierarchical indexes [46, 68].

For each atom A_i, set X_i is computed (line 67) to contain the variables relevant to the rest of the query, and it is used to project out unnecessary variables from the matches of $A_i\sigma$ (line 69) and the partial answer σ' (line 72). Also, each partial answer message $\mathsf{PAR}[i, \sigma, m, \vec{\lambda}]$ includes the multiplicity m of σ, which is combined (line 73) with the multiplicity c of a match ρ of $A_i\sigma$ to obtain the multiplicity m' of σ'.

5.2.3 Backjumping

Index nested loop joins can sometimes perform unnecessary work, as the following example shows.

Example 16.

$$Q_6(x, y_1, y_2, y_3) = \langle x, R, y_1\rangle \wedge \langle x, S, y_2\rangle \wedge \langle x, T, y_3\rangle$$
$$\mathbf{G}_1 = \{\langle a, R, b\rangle, \langle a, S, c_1\rangle, \ldots, \langle a, S, c_k\rangle, \langle d, R, b\rangle, \langle d, S, c_1\rangle, \ldots, \langle d, S, c_k\rangle\} \quad \mathbf{G}_2 = \{\langle d, T, e\rangle\}$$

Let $\sigma_i = \{x \mapsto a, y_1 \mapsto b, y_2 \mapsto c_i\}$. Matching the first two atoms of Q_6 produces partial answer σ_1. Since $\langle x, T, y_3\rangle\sigma_1$ cannot be matched, the unoptimised algorithm tries σ_2. However, σ_2 differs from σ_1 only on y_2, which does not occur in $\langle x, T, y_3\rangle$, so $\langle x, T, y_3\rangle\sigma_2$ still cannot be matched. Thus, our algorithm explores all σ_i in vain.

This inefficiency can be avoided by observing that, when matching $\langle x, T, y_3\rangle\sigma_1$ fails, we must change the value of x to have a chance of a match; thus, we can *backjump* to the first atom and continue evaluation there. To generalise this idea, our algorithm computes, for each variable x, the index I_x of the atom in the left-to-right plan that produces a binding for x (line 63). Moreover, for each atom A_i, it also computes the index B_i of the closest preceding atom that binds a variable in A_i (line 66). Then, if $A_i\sigma$ cannot be matched (line 70), function MatchAtom returns the index of the atom where query evaluation should continue (line 70), which is used to unwind the recursive calls to the desired level (line 86).

Our example also illustrates a subtlety that arises in a distributed setting. Let $\nu_i = \{x \mapsto d, y_1 \mapsto b, y_2 \mapsto c_i\}$. Server 1 will compute ν_1 and send it to server 2, which then computes the answer $\nu_1 \cup \{y_3 \mapsto e\}$. Now ν_1 cannot be extended on server 1, but backjumping to the first atom misses ν_j for $j > 1$, each leading to an answer $\nu_j \cup \{y_3 \mapsto e\}$. Intuitively, backjumping can occur only if an atom cannot be matched in the entire graph, and not just on the current server. To remedy this, backjumping is initiated only if $\textsc{Occurs}(A_i\sigma, \vec{\lambda})$ determines that no server other than the current one can match $A_i\sigma$ (line 70); as mentioned in Section 5.2.1, this is why the recursive call to MatchAtom in line 85 uses $\vec{\lambda}$ instead of $\vec{\lambda}'$. To avoid redundant checks, it is performed only if backjumping would have an effect (i.e., if $B_i \neq i - 1$). With this change, our algorithm backjumps after considering σ_1 (resource a occurs only on server 1), but not after ν_1 (resource d also occurs on server 2).

5.2.4 Early Pruning

Occurrences can be used to further reduce the work during matching, as the following example shows.

Example 17.

$$Q_7(x) = \langle x, R, y\rangle \wedge \langle y, S, z\rangle \wedge \langle z, T, x\rangle$$

Assume now that server k matches $\langle x, R, y\rangle$ via assignment ρ. Since x occurs in $\langle z, T, x\rangle$ in object position, $\rho(x)$ must occur in G in object position for ρ to be extended to an answer. Also, $\langle x, R, y\rangle\rho \in \mathbf{G}_k$ ensures $\rho(x) \in \mathsf{voc}(\mathbf{G}_k)$, so $\mu_{k,o}(\rho(x))$ is defined; hence, server k can check whether $\rho(x)$ occurs on some partition element in object position.

Our algorithm thus computes, for each variable x, the set of positions D_x at which x occurs in the query (line 64). Then, a match ρ of $A_i\sigma$ is skipped (line 71)

if a variable x matched by ρ and a position $\pi \in D_x$ exist such that $\rho(x)$ does not occur in G in position π (lines 89–91).

5.2.5 Parallelism

The processing at each server is done in parallel, as in line 14, where multiple message processing threads are initiated after a new query is received. This is in addition to the thread that services the network buffers. Each message processing thread repeatedly calls GetFromQueue(1), which pulls a message from the message queues and processes it according to Algorithm 2. Any number of message processing threads can be initiated as they can pull messages from the queue of any stage, thus the number of threads is not tied to the size of the query. This provides more flexibility in how parallelism is used. Although each thread maintains their own physical iterators and arguments buffers for calls to MatchAtom, there are some shared resources; these are the message queues and the variables P_i, R_i, N_i, and F_i. To ensure thread safety, in the message queues, a pointer is maintained to the next message, which is modified with an atomic operation when a thread takes a message. Each variable P_i, R_i, and N_i also gets modified by atomic operations. Moreover, F_i, which is used to signify that stage i is complete, is modified using a compare-and-swap operation in line 29. This ensures that only one thread will proceed into the body of the `if` statement, so that termination messages are only sent once.

In addition to the intra-query parallelism described above, our approach allows for inter-query parallelism, i.e. executing multiple queries in parallel. The details of inter-query parallelism were not described in the algorithms in order to aid clarity, however, it is quite simple. Each query is assigned a unique ID by the query coordinator, and all messages for that query contain that ID. Then, queries can be executed in parallel without interference. Furthermore, queries run in parallel can be initiated from different servers; any given server can act as the query coordinator.

5.3 Correctness

Theorem 2. If Algorithms 1, 2, and 4 are applied to a strict partition **G** of an RDF graph G distributed over a cluster C of servers where each server has $n+1$ finite message queues,

1. servers terminate after sending $n|C|^2$ FIN messages,
2. the coordinator for Q correctly outputs $\mathsf{ans}(Q, G)$, and
3. at most $O(n^2)$ memory is needed per server thread.

Proof of Claims 1 and 3. We assume that the servers evolve over discrete time instants $1, 2, \ldots$ where, at each time t, one thread of one server performs an action. We say that server k is *finished* for stage $i \geq 1$ at time t if $P_i = R_i$ and $N_i = |C|$ hold at server k after the action at time t. Also, to unify the case analyses, we define all servers as finished for stage $i = 0$ at each time $t \geq 0$, and we define all servers as not finished for each stage $i \geq 1$ at $t = 0$. When server k becomes finished for stage i at time t, SWAP in line 29 ensures that the server sends precisely one $\mathsf{FIN}[i+1, S_{i+1,\ell}]$ message to each server $\ell \in C$ if $i < n$, or to the coordinator if $i = n$. We next prove that, for each $t \geq 1$,

(a). if a server is finished for stage i at time t, then all servers are finished for stage $i-1$ at time $t-1$,

(b). a server finished for stage i at time t does not have a queued PAR/ANS message for stage i and it is not processing such a message at time t, and

(c). if a server is finished for stage i at time $t-1$, the server remains finished for stage i at time t.

The proof is by induction on t, where the base case and inductive step are the same. Consider a time $t \geq 1$ such that all properties hold for $t-1$. Assume that

server k is finished for stage i at time t. Server k increments its counter N_i in line 25 once for each FIN message received for stage i, and each server sends to server k just one FIN message per stage; hence, $N_i = |C|$ ensures that all servers have sent their FIN messages for stage $i-1$ before time t (if $i \geq 1$), so all servers are finished for stage $i-1$ at time $t-1$, as required for property (a). Moreover, the inductive hypothesis for properties (a) and (c) ensures that all servers are finished for each stage $j < i$ at time $t-1$. Thus, by property (b), no server is sending a PAR/ANS message for stage $j < i$ to server k at time $t-1$, so server k is not retrieving such a message from a message queue at time t. Lines 43 and 26 ensure that counter R_i at server k contains the number of received PAR/ANS messages for stage i, and counter P_i at server k is incremented in line 18 or 22 whenever such a message is processed; hence, $P_i = R_i$ implies that server k has no queued PAR/ANS messages for stage i and is not processing such messages, as required for property (b). Finally, if server k is finished for stage i at time $t-1$, the observations made this far ensure that counters N_i, R_i, and P_i of server k remain unchanged at time t, so server k is finished for stage i at time t, as required property (c).

To complete the proof, each $\mathbf{G}_k$ is finite so, in each call to MatchAtom, server k passes through the loop in lines 71–86 a finite number of times and thus produces finitely many PAR/ANS messages. Moreover, at each point in time, at least one queue in the cluster contains a message with the highest stage index, and this message is eventually processed either in the message processing thread or in line 41 of Algorithm 2. Thus, at each time, at least one server makes progress, so all messages are processed eventually; hence, all servers will eventually become finished for stage n, and the coordinator will become finished for stage $n+1$. Each server sends $|C|$ FIN messages per stage, so the total number of such messages is $n|C|^2$. Finally, recursion depth of each thread is n, and each recursion level can be implemented using n iterators that explore set E in line 69 'on the fly', so each thread requires (n^2) memory. □

Proof of Claim 2. Our algorithms are clearly sound (i.e., $\sigma \in \mathsf{ans}(Q, G)$ holds for each $\langle \sigma, m \rangle$ that is output), so we next prove completeness. Let I_x, D_x, B_i, and X_i be as in lines 62–67 of Algorithm 4; let $\vec{A} = A_1, \ldots, A_n$ be the reordered atoms of $Q(\vec{x})$ from line 2 of Algorithm 1; for each $1 \leq i \leq n$, let $K_i = X_i \cap \bigcup_{j=1}^{i-1} \mathsf{vars}(A_j)$, let $\vec{x}_i = \vec{x} \setminus K_i$, and let $Q_i(\vec{x}_i) = A_i \wedge \cdots \wedge A_n$; and let $\vec{\mu} = \mu_s, \mu_p, \mu_o$ be the vector of occurrence mappings from Section 5.1.2. We say that a vector $\vec{\lambda} = \lambda_s, \lambda_p, \lambda_o$ of partial occurrence mappings is *consistent* with $\vec{\mu}$ if $\lambda_\pi \subseteq \mu_\pi$ for each $\pi \in \Pi$ (i.e., each λ_π coincides with μ_π on the common resources). We also say that a server k *can match* an atom A if an assignment ρ exists such that $A\rho \in \mathbf{G}_k$. The following property clearly holds for each atom A and vector $\vec{\lambda}$ consistent with $\vec{\mu}$:

($*$) $k \in \textsc{Occurs}(A, \vec{\lambda})$ holds for each server $k \in C$ that can match A.

Next, we show that, for each $1 \leq i \leq n$, assignment σ with $\mathsf{dom}(\sigma) = K_i$, positive integer m, vector $\vec{\lambda}$ consistent with $\vec{\mu}$, and $k \in C$, if $\textsc{MatchAtom}(i, \sigma, m, \vec{\lambda})$ is called on server k and the call returns $v < i - 1$, then there exists $j \geq i$ such that $B_j = v$ and $A_j\sigma\rho \notin G$ for all assignments ρ. The proof is by induction on recursion depth. For the base case, assume that v is returned in line 70 and consider an arbitrary assignment ρ. Then, $\textsc{Occurs}(A_i\sigma, \vec{\lambda}) \subseteq \{k\}$ and ($*$) ensure $A_i\sigma\rho \notin G \setminus \mathbf{G}_k$, and $E = \emptyset$ ensures $A_i\sigma\rho \notin \mathbf{G}_k$; thus, the claim holds for $j = i$. For the induction step, assume that v is returned in line 86 after a recursive call in line 85 for assignment σ'. Due to $v < i$ and the induction hypothesis, there exists $j \geq i + 1 > i$ such that $B_j = v$ and $A_j\sigma'\rho \notin G$ for all assignments ρ. But then, the definition of B_i and I_x ensure $A_j\sigma' = A_j\sigma$, so the inductive claim holds for j.

Finally, we show that, for each $1 \leq i \leq n$, assignment σ with $\mathsf{dom}(\sigma) = K_i$, positive integer m, and vectors $\vec{\lambda}_k$ for $k \in C$ consistent with $\vec{\mu}$, if $\textsc{MatchAtom}(i, \sigma, m, \vec{\lambda}_k)$ is called on each server k that can match $A_i\sigma$, then, for each $\nu \in \mathsf{ans}(Q_i\sigma, G)$ with multiplicity w, the coordinator outputs tuples $\langle \sigma \cup \nu, p_1 \rangle, \ldots, \langle \sigma \cup \nu, p_r \rangle$ such that $p_1 + \cdots + p_r = m \cdot w$. This property proves Claim 2 of Theorem 2 since line 5 of

Algorithm 1 calls MatchAtom (via line 17) for $i = 1$, $\sigma = \emptyset$, $m = 1$, and $\lambda_k = \vec{\emptyset}$. The proof is by 'reverse' induction on i going from n down to 1.

Many observations are the same for the base case and the inductive step, so we consider them first. Let $\rho = \nu|_{X_i}$, for $k \in C$ let c_k be the number of distinct extensions ρ' of ρ by the variables in $A_i\rho$ where $A_i\sigma\rho' \in \mathbf{G}_k$, let c be the number of such ρ' where $A_i\sigma\rho' \in G$, let $\sigma' = \sigma \cup \rho$, and let $\nu' = \nu \setminus \rho$ (i.e., ν' contains the mappings of all variables in ν not covered by ρ). Since $\mathbf{G}$ is strict, we have $c = \sum_k c_k$. It should be clear that $\nu' \in \mathsf{ans}(Q_{i+1}\sigma', G)$, and that the multiplicity w' of ν' satisfies $w = w' \cdot c$. Now consider an arbitrary server $k \in C$ with $c_k \neq 0$; thus, server k can match atom $A_i\sigma$ so MatchAtom is called there. By the definition of Evaluate, we have $\langle \rho, c_k \rangle \in E$ in line 69. Thus, $E \neq \emptyset$ so function MatchAtom does not return in line 70. For each variable $x \in \mathsf{dom}(\rho)$ and position $\pi \in D_x$, the definition of D_x ensures that atom A_j exists such that $j > I_x$ and $x = \mathsf{term}_\pi(A_j)$; but then, ν' being an answer to $Q_{i+1}\sigma'$ ensures $\mu_\pi(\rho(x)) \neq \emptyset$; moreover, $x \in \mathsf{dom}(\rho)$ ensures $\rho(x) \in \mathsf{voc}(\mathbf{G}_k)$, so $\mu_{k,\pi}$ is defined and it satisfies $\mu_{k,\pi}(\rho(x)) \neq \emptyset$ and CanPrune(ρ) returns *false* in line 71. Finally, if a recursive call in line 85 were to return a number smaller than i, then some $j > i$ would exist such that $A_j\sigma\xi \notin G$ for all assignments ξ, so ν could not be an answer to $Q_i\sigma$. Hence, MatchAtom does not return in line 86, so ρ, c_k, and σ' are considered at some point in the body of the loop in lines 71–86 on server k. We next show that all servers jointly produce the required answers.

For the base case $i = n$, server k outputs σ' with multiplicity $m \cdot c_k$ in line 76 or via line 78 and lines 20–21 of Algorithm 2. The multiplicity of σ' aggregated over all servers is $\sum_k m \cdot c_k = m \cdot \sum_k c_k = m \cdot w$, as required.

Now consider the induction step. Vector $\vec{\lambda}'$ is clearly consistent with $\vec{\mu}$. Property $(*)$ ensures that the set L in line 80 contains each server ℓ that can match $A_{i+1}\sigma'$; server k sends to each such ℓ a partial answer message in line 83; and, by Claim 1, ℓ eventually processes the message in lines 16–18 of Algorithm 2. By the inductive

assumption, then σ' is output with aggregated multiplicity $m \cdot c_k \cdot w'$. Finally, MatchAtom is called on each server k that can match $A_i\sigma$, so the multiplicity of σ' aggregated over all servers is $\sum_k m \cdot c_k \cdot w' = m \cdot (\sum_k c_k) \cdot w' = m \cdot w$, as required.

□

5.4 Conclusion

In this chapter we presented a novel approach to distributed query evaluation based on *dynamic data exchange* that provides fully asynchronous execution and network communication, as well as providing strong memory guarantees to ensure query evaluation will complete successfully, even in low-memory conditions. To facilitate this fully asynchronous approach, we also introduced a novel lightweight termination detection algorithm. Furthermore, we presented several optimisations to our approach such as partial occurrences, which reduce the storage overhead to ensure horizontal scalability, backjumping, which reduces the amount of redundant computation done, and early pruning which prunes partial answers that cannot be extended elsewhere in the cluster. Finally, we provided a full proof of correctness for our approach.

6
Query Planning

RDF stores typically use a *query planner* to identify a query plan that reduces the time or resources required for query evaluation. A planner usually comprises (i) a *query cardinality estimator* that uses statistics about the data to estimate the number of answers to the query or its subparts, (ii) a *cost model* that combines these estimates into a numeric measure of the time and resources needed, and (iii) a *query planning algorithm* that finds, or at least approximates, a plan with the least cost. We can apply these principles in our setting, but distributed processing raises several important issues.

First, the cost of processing at a server is determined by the number of messages that it produces as well as the number of partial answers that it considers. Thus, in Section 6.1.1 we present a novel technique that can estimate these numbers by reusing any query cardinality estimator.

Second, combining the costs of processing and communication of all servers is not trivial since servers (and sometimes even communication) operate in parallel. Thus, in Section 6.1 we discuss how to define the plan cost so that it approximates the system's performance.

For the rest of this section, we fix a query $Q(\vec{x})$ that is to be evaluated over a strict partition $\mathbf{G}$ of an RDF graph G. Since we use index nested loop joins, a plan for Q is a reordering $\vec{A} = A_1, \ldots, A_n$ of the atoms of Q. Also, we consider the MatchAtom variant from Algorithm 3: capturing the optimisations from Section 5.2 would not significantly affect plan quality.

6.1 Cost Model

Consider a call to MatchAtom for stage i of plan $\vec{A}$ on server k. We present a query $\mathsf{P}_{\vec{A},i,k}$ that, on an RDF graph G' obtained by extending G with information about resource occurrences, returns precisely the partial answers σ' considered in line 46 so the number of such σ' determines the number of passes through the loop in lines 45–55. We also present a query $\mathsf{S}_{\vec{A},i,k,\ell}$ whose number of answers, when evaluated over G', is equal to the number of messages sent from server k to server ℓ in line 51 or 54.

These numbers are the basic building blocks of our cost model. Since $\mathsf{P}_{\vec{A},i,k}$ and $\mathsf{S}_{\vec{A},i,k,\ell}$ are just CQs, we can estimate the numbers of their answers using an arbitrary cardinality estimator (which is likely to require statistics about G'). We can then use these estimates to build a model of the cost at each server and then the total cost of the query plan.

6.1.1 Counting Partial Answers and Sent Messages

We first define G'. Let occurs_s, occurs_p, occurs_o, and srv_k for $k \in C$ be fresh resources not occurring in G. Then, G' extends G by triples $\langle r, \mathsf{occurs}_\pi, \mathsf{srv}_k \rangle$ for each position $\pi \in \Pi$, resource $r \in \mathsf{voc}_\pi(G)$, and server $k \in \mu_\pi(r)$. Such triples encode resource occurrences, but we shall need triple occurrences as well. For simplicity, we first assume that all triples containing a resource r in their subject are assigned to the same partition element $\mathbf{G}_k$; thus, $\mu_s(r) = \{k\}$—that is, r occurs in the subject position only on server k. Such triple assignment is practically beneficial as it is

allows answering subject–subject joins without any communication. We discuss below ways around this restriction.

Now let $\mathsf{P}_{\vec{A},i,k}$ be the following query, where $\vec{x}_i$ are all variables occurring in $A_1 \wedge \cdots \wedge A_i$:

$$\mathsf{P}_{\vec{A},i,k}(\vec{x}_i) = A_1 \wedge \cdots \wedge A_i \wedge \langle \mathsf{term}_s(A_i), \mathsf{occurs}_s, \mathsf{srv}_k \rangle \tag{6.1}$$

Evaluating $A_1 \wedge \cdots \wedge A_i$ on G clearly produces each assignment σ' considered in stage i in line 46 of Algorithm 3 on any server. Moreover, by our assumption on data partitioning, the last atom of $\mathsf{P}_{\vec{A},i,k}$ is true whenever atom A_i is matched in $\mathbf{G}_k$. Thus, σ' is an answer to $\mathsf{P}_{\vec{A},i,k}$ on G' if and only if σ' is considered in stage i on server k.

Also, for $i < n$, let $\mathsf{S}_{\vec{A},i,k,\ell}$ be as follows, where set B contains each *bound* position π of A_{i+i}—that is, $\pi \in B$ if and only if $\mathsf{term}_\pi(A_{i+1})$ is a resource or a variable occurring in $\vec{x}_i$:

$$\mathsf{S}_{\vec{A},i,k,\ell}(\vec{x}_i) = \mathsf{P}_{\vec{A},i,k} \wedge \bigwedge_{\pi \in B} \langle \mathsf{term}_\pi(A_{i+1}), \mathsf{occurs}_\pi, \mathsf{srv}_\ell \rangle \tag{6.2}$$

For each σ', server k sends a PAR message for stage $i+1$ to server ℓ if all resources of $A_{i+1}\sigma'$ occur on server ℓ. Since B contains each position at which $A_{i+1}\sigma'$ contains a resource, σ' is an answer to $\mathsf{S}_{\vec{A},i,k,\ell}$ on G' iff server k sends a PAR message for stage $i+1$ containing σ' to server ℓ. For $i = n$, the ANS messages that server k sends to the coordinator (if k is not the coordinator) are determined by just $\mathsf{P}_{\vec{A},n,k}$.

We now discuss how to handle cases when triples are not assigned to partition elements by subject. Query $\mathsf{S}_{\vec{A},i,k,\ell}$ then remains unchanged since it uses resource occurrences only, but $\mathsf{P}_{\vec{A},i,k}$ needs to be adapted. Intuitively, G' must then also capture locations of triples, which can be done in (at least) one of two ways. First, we can use reification: for each triple $\langle t_s, t_p, t_o \rangle \in \mathbf{G}_\ell$, we introduce a fresh resource t and transform the triple as $\langle t, \mathit{rdf{:}subject}, t_s \rangle$, $\langle t, \mathit{rdf{:}predicate}, t_p \rangle$, and $\langle t, \mathit{rdf{:}object}, t_o \rangle$, and we further record the location of the triple by $\langle t, \mathsf{occurs}, \mathsf{srv}_\ell \rangle$. We also transform the atoms of (6.1) accordingly, and we require atom A_i to be

matched on server k. Second, we can use *quads*—an extension of RDF where basic data units are quadruples of resources. Thus, we transform each $\langle t_s, t_p, t_o \rangle \in \mathbf{G}_\ell$ into $\langle t_s, t_p, t_o, \mathsf{srv}_\ell \rangle$, and we adapt the atoms of (6.1) accordingly.

We next use the cardinalities of $\mathsf{P}_{\vec{A},i,k}$ and $\mathsf{S}_{\vec{A},i,k,\ell}$ to determine the cost of a plan $\vec{A}$.

6.1.2 Cost at Each Server

We first consider the processing at each server in the cluster. Capturing a server's behaviour on $\vec{A}$ is very challenging, so we make several simplifying assumptions. First, we assume that all servers in C have the same numbers of threads, each taking an equal share of the server's workload. This will hold if local processing can be parallelised, but a discussion of the issues involved is out of scope here. Second, we assume that a server never waits for messages—that is, a message is available whenever a server's thread is idle. This can be expected to hold whenever the number of sent messages is not very small. Third, we assume that the queues of all servers are large enough so that line 40 of Algorithm 2 always succeeds. As long as the queues are reasonably large, this should not significantly affect query planning: flow control will be needed only on plans producing many partial answers, but the cost of such plans will anyway be high. Fourth, we assume that retrieving each assignment ρ in line 45 of Algorithm 3 requires a fixed amount of time. This can be expected to hold in most implementations that match atoms using indexes.

With these assumptions in mind, the cost of local processing at server k is then proportional to the number of passes through the loop in lines 45–55, which is given by

$$\mathsf{local}_k(\vec{A}) = \sum_{i=1}^{n} |\mathsf{P}_{\vec{A},i,k}|_{G'}. \tag{6.3}$$

To estimate the amount of data sent by server k, let M_i be average the size of a PAR/ANS message for stage $1 \leq i \leq n+1$. This should be easy given that the message size depends on the number of variables being sent, which is determined

by i; in fact, we can derive M_i from the number of variables in X_i that are bound, where X_i is defined in line 67 of Algorithm 4. Then, if server k is the coordinator, it will send a total of

$$\mathsf{send}_k(\vec{A}) = \sum_{i=1}^{n-1} \left(\mathsf{M}_{i+1} \cdot \sum_{\ell \in C \setminus \{k\}} |\mathsf{S}_{\vec{A},i,k,\ell}|_{G'} \right) \tag{6.4}$$

bytes to other servers. If k is not the coordinator, we further extend (6.4) by $\mathsf{M}_{n+1} \cdot |\mathsf{P}_{\vec{A},i,k}|_{G'}$ to also account for the ANS messages that server k sends to the coordinator.

6.1.3 Combining the Cost of all Servers

The cost of a plan $\vec{A}$ must reflect the fact that servers are largely independent and work in parallel. Ganguly et al. [21] present a general framework for query planning that can be applied in such a setting. They generalise the cost of a plan $\vec{A}$ to be a vector $\vec{V}_{\vec{A}}$ and a relation that provides a partial order of the plans, and they show that this definition is incompatible with standard dynamic programming algorithms. So far we have modelled the cost at a server to be proportional to the amount of time it would take to process the query. When combining the cost of all servers, we will then aim to minimise the amount of time the system as a whole takes to answer the query by choosing the plan with the minimum maximum cost across all server. There are many other objectives a query planner could have, such as minimising the total work done across the system, which would maximise the throughput if it were to receive a stream of queries. We do not consider other such objectives in this work, but note our approach could be easily adapted to do so.

Finally, they present a variant of the dynamic programming approach to query planning that can handle partially ordered cost models.

To apply this well-known technique, we must define $\vec{V}_{\vec{A}}$, which requires addressing the following two issues.

First, $\mathsf{local}_k(\vec{A})$ and $\mathsf{send}_k(\vec{A})$ are incomparable: the former is the number of passes through a loop, and the latter is the number of bytes. We therefore scale $\mathsf{send}_k(\vec{A})$ by a factor f so that both numbers reflect processing time. To select f, we can measure the average time for processing one loop iteration and compare it with the cost of network communication. For example, if each pass through the loop takes about 10 μs, all servers use just one thread, the cluster uses gigabit Ethernet, and we disregard network congestion, then we can take $f = 0.08$ as the ratio of the times required to send one byte and to process one loop iteration.

Second, we must combine $\mathsf{local}_k(\vec{A})$ and $\mathsf{send}_k(\vec{A})$ as appropriate for the implementation at hand. We see two main possibilities for this. First, if a call to PutIntoQueue is likely to block the sending thread (e.g., if sent messages are written directly onto a TCP connection), then the two costs should be added; hence, we define $\vec{V}_{\vec{A}}$ as containing $\mathsf{local}_k(\vec{A}) + f \cdot \mathsf{send}_k(\vec{A})$ for each server $k \in C$. Second, if PutIntoQueue just copies messages into a large queue that is processed in the background, then message sending is just another parallel tasks; hence, we define $\vec{V}_{\vec{A}}$ as containing both $\mathsf{local}_k(\vec{A})$ and $f \cdot \mathsf{send}_k(\vec{A})$ for each $k \in C$.

6.2 Dynamic Programming

With the cost model in place, the final task is to the find the lowest cost plan. A common strategy is to use dynamic programming, which sees widespread use in commercial systems, pioneering in IBM's System R [60]. This approach iteratively builds larger plans by combining two previously built plans, keeping only the lowest cost plan for each set of atoms. The advantage of this approach is the guarantee of finding the lowest cost plan. The disadvantage is it has exponential time and space complexity, although in practice this is not prohibitive.

This method is shown in Algorithm 5, which will build left-deep plans only, as this is the only type of plan we consider in Chapter 5. A bottom-up approach is

Algorithm 5 Basic Dynamic Programming Algorithm for Left-Deep Plans

$Q = A_1 \wedge \cdots \wedge A_n$

```
98: for i from 1 to n do
99:     optPlan({A_i}) = planAtom(A_i)
100: for i from 2 to n do
101:     for all T ⊆ {A_1, ..., A_n} where |T| = i do
102:         for all A ∈ T do
103:             plan := joinPlan(optPlan(T \ A), optPlan(A))
104:             if optPlan(T) is null or plan ≤_cost optPlan(T) then
105:                 optPlan(T) := plan
106: return optPlan({A_1, ..., A_n})
```

used, by first building plans for single atoms (line 99). Next, subplans of increasing size are built by considering all plans that extend plans from the previous iteration by a single atom; this is done by joining plans from the previous iteration with plans from the first iteration (line 103). Each plan created in this way is pruned if a lower cost plan has already been built from the same set of atoms (lines 104-105). This step relies on a relation $\leq_{cost}$ which provides a total ordering on plans according to some cost function on plans. As plans are pruned at the earliest point, the search space of the algorithm is greatly reduced, as no subsequent plans are built from the higher cost plans. Moreover, this pruning step guarantees that the optimal plan for the whole set of atoms is a singleton.

The problem with the standard dynamic programming algorithm for query planning is the cost model for a plan has to produce a total ordering on plans. This is because the algorithm chooses the single lowest cost plan for each subset of the atoms in the query, which is necessary to guarantee the algorithm produces a plan with the lowest cost. As the cost model in 6.1 uses a vector to represent the cost, this is not possible. Furthermore, reducing the cost vector to a single value by taking the maximum value does not solve the problem because of the iterative nature of dynamic programming. That is, extending two subplans of the same set of atoms with the same atom does not imply the resulting plan from the lower

cost subplan will have a lower cost. Thus, the pruning step in lines 104-105 will potentially prune plans in error, and there is no longer a guarantee of finding the lowest cost plan. We illustrate this problem in the example below.

Example 18. *Let* $Q = A_1 \wedge \cdots \wedge A_n$ *and* $\vec{A}_1$ *and* $\vec{A}_2$ *be reorderings of atoms* $A_1, \ldots A_{n-1}$ *with respective cost vectors* $\langle 10, 1, \ldots, 1\rangle$ *and* $\langle 9, \ldots, 9\rangle$*, Furthermore, assume the cost of extending both plans with atom* A_n *has cost* $\langle 0, 5, \ldots, 5\rangle$*. It is clear that plan* A_1 *leads to the lowest cost plan; however, this plan would be pruned if the maximum was taken.*

Ganguly et al. [21] present a generalisation of the dynamic programming approach for partial orders, which can be used in the case where costs are vectors. First, we have to redefine the relation $\leq_{cost}$ to produce a partial order over the plans.

Definition 9 ([21])**.** *Given two plans* $plan_x$ *and* $plan_y$ *with associated cost vectors,* $\langle x_1, \ldots, x_m\rangle$ *and* $\langle y_1, \ldots, y_m\rangle$*, the relation* $\leq_{cost}$ *is defined as* $plan_x \leq_{cost} plan_y$ *if and only if* $x_i \leq y_i$ *for all* i *in* $[1..m]$*.*

Using this redefined relation, pruning of plans can happen the same way as previously. However, for a given set of atoms of a query, it is possible that more than one plan is left after pruning, as we now only have a partial ordering on plans. This increases the search space for plans, but still guarantees the lowest cost plan will be found. The modifications necessary to Algorithm 5 are shown in Algorithm 6. *optPlan* is now a set of plans, and each one needs to be extended (line 112). Furthermore, when pruning, all plans with higher cost according to the new $\leq_{cost}$ relation must be pruned (lines 114-116). When the final plans for Q have been generated, the lowest cost plan is then selected with the function bestPlan in line 118. The relation $\leq_{cost}$ cannot be used here because it does not produce an ordering of the plans. However, the lowest cost plan, i.e. the one that produces the minimum overall execution time, is the with the minimum maximum element of its cost vector.

Algorithm 6 Generalised Dynamic Programming Algorithm for Left-Deep Plans

$Q = A_1 \wedge \cdots \wedge A_n$

for i from 1 to n **do**
  $optPlan(\{A_i\}) = planAtom(A_i)$
for i from 2 to n **do**
  for all $T \subseteq \{A_1, \ldots, A_n\}$ **where** $|T| = i$ **do**
    for all $A \in T$ **do**
      for all $leftPlan \in optPlan(T \setminus A)$ **do**
        $plan := joinPlan(leftPlan, optPlan(A))$
        if $\nexists oldPlan \in optPlan(T)$ **where** $oldPlan \leq_{cost} plan$ **then**
          for all $oldPlan \in optPlan(T)$ **where** $plan \leq_{cost} oldPlan$ **do**
            $optPlan(T) := optPlan(T) \setminus oldPlan$
          $optPlan(T) := optPlan(T) \cup plan$
return $bestPlan(optPlan(\{A_1, \ldots, A_n\}))$

6.3 Conclusion

In this chapter we presented a novel approach to query planning for query evaluation that uses dynamic data exchange by showing how existing query cardinality estimators and generalised dynamic programming approaches can be used and adapted to accurately model the costs of query evaluation. This was done by creating queries that estimate both the number of intermediate answers produced and the number of partial answers sent between servers. By representing the cost of a query plan as a vector consisting of costs for each machine, we are able to include the inherent parallelism in the cluster in the cost model. Finally, we showed how the lowest cost plan can be found through a generalised version of the commonly used dynamic programming-based algorithm.

7
Data Partitioning

Data partitioning is an important problem in many areas of information system design, such as distributed databases, parallel databases, and multi-threaded systems. Each of these areas are concerned with how data is stored and accessed in systems with multiple devices, components, files, or physical areas. Data partitioning groups data and associates it with particular instances of these locations. Moreover, it can include duplication of data to provide redundancy or improve system performance, at the cost of storage overhead.

In Chapter 3 we surveyed the data partitioning techniques used by existing distributed systems, where the primary goal was often to reduce the number of data exchange operators used in typical queries for their respective domains. This often included significant levels of duplication, such as n-hop duplication or both subject and object hashing. The problem with many of these schemes is only queries their schemes were tailored to were guaranteed to be answered efficienctly, and otherwise inefficient, data-communication heavy evaluation strategies are required. Furthermore, the query evaluation strategies do not take advantage of the serendipitous data locality: data that is located on the same server, but was not guaranteed to be on the same server by the partitioning scheme.

Our query evaluation strategy based on dynamic data exchange, as described in Chapter 5 does not rely on, or even use, guarantees provided by the partitioning scheme as it makes decisions on where and when to send data at run-time rather than at compile-time. This provides greater flexibility to the partitioning scheme, and hence we devise a novel data partitioning strategy to complement dynamic data exchange that (i) aims to maximise the number of local answers on common queries, (ii) does not duplicate triples, and (iii) produces partitions balanced in the number of triples. Our approach is based on *weight graph partitioning* and an extension of the ideas presented in Chapter 4. The main differences to the earlier approach is not being constrained to partitioning the vocabulary of the graph, not needing wildcard resources, and the use of weighted graph partitioning to balance the number of triples between partitions.

7.1 Graph Partitioning

A feature of the RDF data model is it can be represented as a graph; given a triple $\langle s, p, o \rangle$, it can be represented as two vertices labelled s and o, with a directed edge from s to o with label p. This naturally gives rise to graph partitioning of RDF data. Graph partitioning has been extensively studied as its own field, outside of information systems, and many approaches and implementations exist [5, 15, 40, 63]. The most common formulation of the problem is *min-cut* partitioning, which partitions the vertices of a graph such that the number of "cut" edges connecting vertices in different partition elements is minimised. This problem is NP-complete. We formalise this problem below.

Definition 10 (**k-way Graph Partitioning**). *Given a graph $G = (V, E)$, partition V into k subsets $V_1, \ldots, V_k$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $\bigcup_{i=1}^{k} V_i = V$ and $|V_i| \approx |V|/k$, while minimising the number of edges $(u, v) \in E$ such that $u \in V_i$, $v \in V_j$, $i \neq j$.*

This definition can be generalised to *weighted* k-way graph partitioning, where a graph has an associated weight relation for vertices and edges. The resulting aim is then to partition the vertices such that the sum of the weights of the vertices in each partition element is the same, while minimising the sum of the weights of the cut edges. The unweighted version is then equivalent to the weighted version with all weights set to one.

It is straightforward to see how this problem can be applied to RDF graphs. This approach attempts to group resources that have a lot of connections and separate those that do not. At an intuitive level, this is beneficial for query evaluation; a query is likely to ask about resources that are connected in some way. Hence, grouping resources that are connected means you are more likely to find a given answer on a single partition element. Compare this to hash partitioning, where given two triples and a partition of size k, the probability those triples are placed on the same machine is $1/k$.

7.2 Weighted RDF Graph Partitioning

Our approach consists of three steps; (i) we transform the RDF graph into an undirected, weighted graph, pruning some edges to improve the quality of the partitioning, (ii) apply weighted graph partitioning to the resulting graph, which produces a mapping from vertices to partition elements, and (iii) reconstruct each partition element as RDF graphs.

We proceed in three steps. First, we transform G into an undirected weighted graph (V, E, w). We define $V = \mathsf{voc}_s(G)$—that is, vertices are resources occurring in G in the subject position. We add to E an undirected edge $\{s, o\}$ for each triple $\langle s, p, o\rangle \in G$ such that $p \neq \mathit{rdf{:}type}$, $o \in \mathsf{voc}_s(G)$, and o is not a literal (e.g., a string or an integer). Finally, we set the weight $w(r)$ of each resource $r \in V$ to the number of triples in G that contain r in the subject position. Ignoring *rdf*:*type* and triples

containing literals avoids introducing vertices that correspond to classes and literals; such vertices often have a large numbers of connections, and this can reduce the effectiveness of graph partitioning algorithms. As we discuss below, this does not affect the performance of our query answering approach on common queries.

Second, we apply *weighted graph partitioning* [40] to (V, E, w): we compute a function $\tau : V \to C$ such that (i) the number of edges spanning partitions is minimised, while (ii) the sum of the weights of the vertices assigned to each partition is approximately the same for all partitions.

Third, we compute each partition element by assigning triples based on subject—that is, we assign each triple $\langle s, p, o\rangle \in G$ to partition element $\mathbf{G}_{\tau(s)}$. Note that this ensures no duplication between partition elements.

Example 19. *Let G be the graph consisting of the following triples in (7.1), to be partitioned across two servers, shown graphically in Figure 7.1a.*

$$\begin{aligned} G = \{\ & \langle b, R, d\rangle, \langle c, R, a\rangle, \langle c, R, d\rangle, \langle d, R, e\rangle, \langle e, R, f\rangle, \langle g, R, f\rangle, \\ & \langle h, R, e\rangle, \langle h, R, f\rangle, \langle a, \mathit{rdf{:}type}, s\rangle, \langle c, \mathit{rdf{:}type}, t\rangle, \\ & \langle d, \mathit{rdf{:}type}, s\rangle, \langle f, \mathit{rdf{:}type}, s\rangle, \langle h, \mathit{rdf{:}type}, t\rangle\} \end{aligned} \tag{7.1}$$

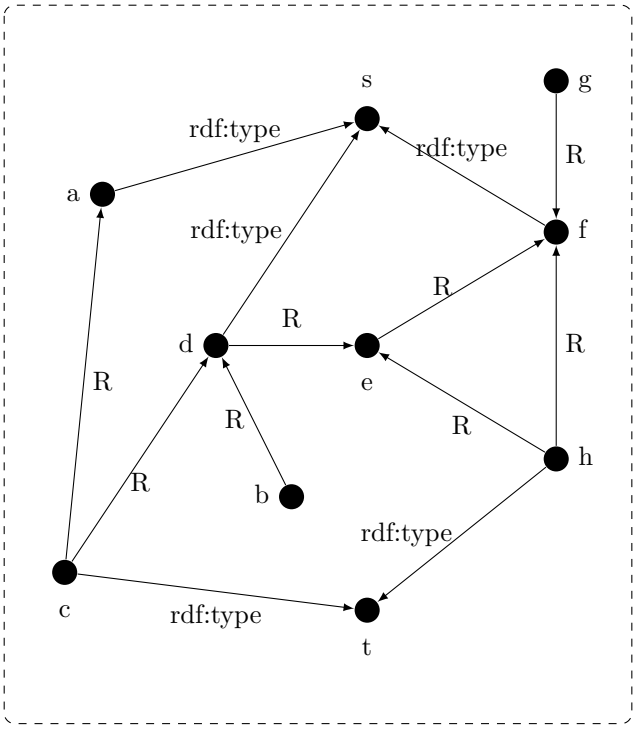
The first step is to transform G to an unweighted graph (V, E, w) to be partitioned. V are the vertices occurring in G in the subject position, $V = \{a, b, c, d, e, f, g, h\}$. E in this case is all the edges whose predicates are not rdf:type, as these are all removed. For each vertex v in V, its weight $w(v)$ is the number of triples in G where v occurs in the subject position. Hence, we have w as in (7.2). This gives the graph as in Figure 7.1b, with the vertices labelled with their weights.

$$\begin{aligned} w = \{\ & a \to 1, b \to 1, c \to 3, d \to 2, \\ & e \to 1, f \to 1, g \to 1, h \to 3\} \end{aligned} \tag{7.2}$$

The second step is to apply weighted graph partitioning to (V, E, w) to compute $\tau : V \to \{1, 2\}$. One reasonable result of this would be the mapping shown in (7.3).

Figure 7.1: Transforming G to an undirected, weighted graph after pruning

(a) G



(b) G

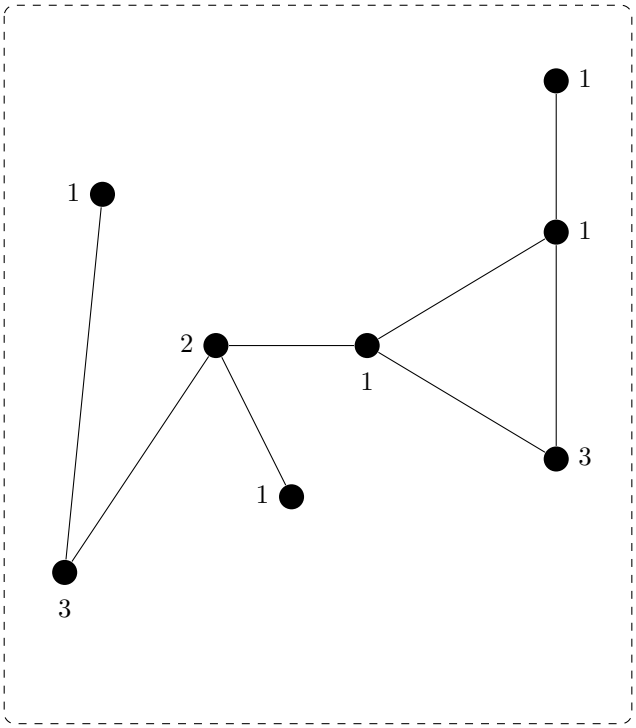
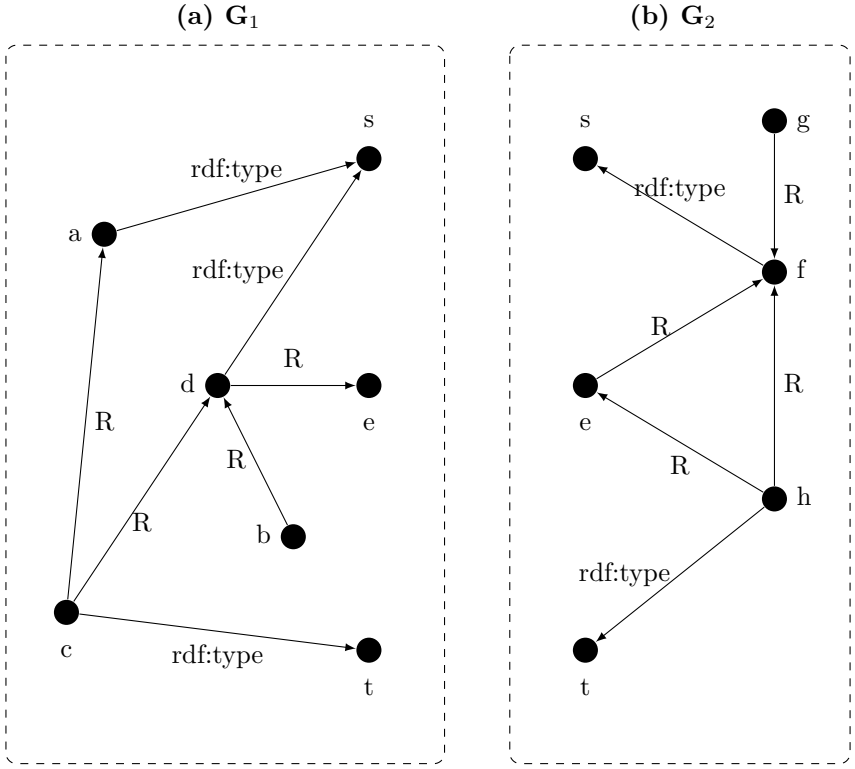


Figure 7.2: Example of our partitioning scheme



$$\tau = \{\ a \to 1, b \to 1, c \to 1, d \to 1, \\ e \to 2, f \to 2, g \to 2, h \to 2\} \tag{7.3}$$

Finally, the third step is to build each partition element $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *by assigning each triple* $\langle s, p, o\rangle \in G$ *to* $\mathbf{G}_{\tau(s)}$*. This gives the partitions in (7.4), shown graphically in Figure 7.2.*

$$\begin{aligned} \mathbf{G}_1 = \{\ & \langle b, R, d\rangle, \langle c, R, a\rangle, \langle c, R, d\rangle, \langle d, R, e\rangle, \\ & \langle a, \mathit{rdf{:}type}, s\rangle, \langle c, \mathit{rdf{:}type}, t\rangle\} \\ \mathbf{G}_2 = \{\ & \langle e, R, f\rangle, \langle g, R, f\rangle, \langle h, R, e\rangle, \langle h, R, f\rangle, \\ & \langle d, \mathit{rdf{:}type}, s\rangle, \langle f, \mathit{rdf{:}type}, s\rangle, \langle h, \mathit{rdf{:}type}, t\rangle\} \end{aligned} \tag{7.4}$$

The pruning step of removing classes and literals is an important part of the strategy. Although at first glance removing information about the structure of the graph could seemingly reduce the quality of the partition, it actually provides many benefits. Firstly, partitioning a smaller graph requires less work in what is a

computationally expensive task. Secondly, as the connectedness and complexity of the graph is increased, the quality of the resulting partition typically decreases; pruning high-degree nodes greatly helps to reduce the connectedness of the graph. Furthermore, pruning these nodes means that triples containing them in the object position are more likely to be placed on separate machines. This can be beneficial for workload balancing. Consider the query below (7.5).

$$Q(x, y) = \langle x, R, y \rangle \wedge \langle x, rdf{:}type, a \rangle \tag{7.5}$$

If all resources of type a are located on the same machine, the work will be done almost entirely on that machine, using only a small part of the computing power of the cluster. However, by spreading them across the cluster, the workload becomes more balanced, which could reduce query repsonse times.

The disadvantage of this approach is can affect the performance of joins involving the object of a triple, as it is more likely to cross server boundaries which is less efficient. A study of more than 3 million real-world SPARQL queries revealed that approximately 60% of joins are subject–subject joins, 35% are subject–object joins, and less than 5% are object–object joins [20]. This suggests that this disadvantage is not a significant problem in practice. As we place all triples with the same subject on a single server, we can answer the most common subject–subject joins without any communication.

Finally, the weight $w(r)$ of each vertex r in (V, E, w) determines exactly the number of triples that are added to $\mathbf{G}_{\tau(r)}$ as a consequence of assigning r to partition $\tau(r)$; since weighted graph partitioning balances the sum of the weights of vertices in each partition, the resulting partition elements are balanced in size. As our experiments in Chapter 8 show, our partitions are indeed much more balanced than the ones produced by the existing approaches based on graph partitioning

[28, 32, 51], which is important as it ensures that each server in the cluster uses roughly the same amount of memory.

7.2.1 Implementation

One of the drawbacks of the k-way partitioning problem is it is NP-complete, and although there are efficient heuristics for finding approximate solutions, such as METIS [40], computing a partition of a graph is an expensive operation. Systems such as METIS require the whole graph to be in memory, which can be problematic in the distributed setting where the size of the graph far exceeds the resources of a single server. Compared to RDF graphs, the representation of the graph required by METIS is very compact, as it is unlabelled and undirected. Furthermore, pruning *rdf*:*type* triples and triples containing literals often significantly reduces the size of the graph. Consequently, an RDF graph can be partitioned using our scheme in Section 7.2 on a server that can not load the RDF graph itself into memory. In order to realise this, in the implementation of our partitioning scheme, the RDF graph is transformed into the graph representation for METIS in a streaming fashion, so the RDF graph does not need to be loaded into memory.

However, it could still be the case that even the reduced graph exceeds the resources of a single server, meaning a centralised graph partitioning system cannot be used. In a similar vain to streaming the graph reduction phase, the graph partitioning phase can also be streamed. This problem is known as *streaming graph partitioning* and has seen a lot of research effort in the recent years, with systems such as [64, 66]. These streaming solutions show that in many cases they can produce partitions that are comparable to state-of-the-art centralised systems, such as METIS. This research reinforces the idea that graph partitioning is a practical tool for use in the distributed setting.

7.3 Conclusion

In this chapter we introduced a novel data partitioning scheme based on weighted graph partitioning to complement the query evaluation strategy presented in Chapter 5. Without the need to provide guarantees on data placement, we were able to take a more flexible approach and utilise the strengths of graph partitioning, without the need for duplication strategies, such as n-hop duplication. Furthermore, by using weighted graph partitioning, we are able to produce partitions which are balanced in the number of triples, rather than the number of resources. We will investigate in Chapter 8 whether this approach can improve query evaluation times.

8 Experimental Evaluation

We implemented our algorithms in the RDFox store.[1] The system's core was written in C++, and METIS [40] is used for graph partitioning. The query planner was written in Java as it reuses a recent query cardinality estimator based on graph summarisation [65]. To understand how data partitioning affects the performance, we evaluated RDFox with graph partitioning from Chapter 7 and with subject hashing, which we call RDFox$_{GP}$ and RDFox$_{HP}$, respectively. We compared the performance of query answering of RDFox with state of the art systems, analysed RDFox's scalability, and we also investigated the uniformity of partitions produced by different partitioning strategies.

8.1 Test Datasets

We based our evaluation on two well-known benchmarks. WatDiv v0.6 [4] aims to simulate realistic data and query loads. We used the original 20 query templates classified into four categories: linear (L), star (S), snowflake (F), and complex (C); each template contains at most one parameter that is replaced with a resource

[1] `http://www.cs.ox.ac.uk/isg/tools/RDFox/`

Table 8.1: Number of triples in test datasets

Dataset	Triples
WatDiv-1K	109.11 M
WatDiv-3K	327.38 M
WatDiv-5K	545.80 M
WatDiv-7K	764.18 M
WatDiv-10K	1.09 G
LUBM-10K	1.33 G

from the RDF graph. In addition to these queries, we also used the 18 incremental linear (IL) queries developed by Schätzle et al. [57].

LUBM [27] is a widely used benchmark in the Semantic Web community. It comes with 14 predefined queries, but most of them do not return any results if reasoning is not used. Thus, we instead used seven queries (T1–T7) by Zeng et al. [74] that compensate for the lack of reasoning, and we manually generated three new complex, cyclic queries. LUBM is a synthetic dataset that represents information about a university, its members, and its activities. It is often cited as an unrealistic benchmark as it simply repeats the same structure for each university when scaling the data upwards. This structure makes certain tasks much easier than realistic datasets, such as partitioning, where data about each university should be grouped together. The advantages, however, are firstly that being synthetic, it can be scaled to arbitrary sizes with fine control, as was useful in the following experiments. Secondly, as LUBM sees widespread use in evaluation across the Semantic Web community, it provides a common first step in comparing various systems.

Table 8.1 shows the sizes of our test graphs. All queries we used are available in the literature, apart from three new LUBM queries which are shown in Table 8.2.

8.2 Query Answering Experiments

We compared the time, network communication, and memory use of RDFox with state of the art systems.

Table 8.2: Three new queries for LUBM

N1	SELECT ?P1 ?D1 ?S1 ?U1 WHERE { ?D1 ub:subOrganizationOf ?U1 . ?P1 ub:worksFor ?D1 . ?S1 ub:advisor ?P1 . ?S1 ub:undergraduateDegreeFrom ?U1 }
N2	SELECT ?S1 ?C1 ?P1 ?C2 ?S2 ?C3 WHERE { ?S2 ub:teachingAssistantOf ?C2 . ?P1 ub:teacherOf ?C2 . ?S2 ub:takesCourse ?C3 . ?S1 ub:takesCourse ?C3 . ?S1 ub:takesCourse ?C1 . ?P1 ub:teacherOf ?C1 }
N3	SELECT ?S1 WHERE { ?S2 ub:teachingAssistantOf ?C2 . ?P1 ub:teacherOf ?C2 . ?S2 ub:takesCourse ?C1 . ?P1 ub:teacherOf ?C1 . ?S2 ub:takesCourse ?C3 . ?S1 ub:takesCourse ?C1 . ?S1 ub:takesCourse ?C3 }

8.2.1 Comparison Systems and Test Setting

Amongst the numerous distributed RDF stores, TriAD [28] has been shown to outperform two centralised (i.e., RDF-3X [46] and BitMat [6]) and four distributed stores (i.e., SHARD [53], H-RDF-3X [32], 4store [30], and Trinity.RDF [74]), the MonetDB [62] column store, and 'raw' Apache Hadoop and Spark. We thus used TriAD as our main point of comparison. The system's authors sent us their code and helped us put it to use. Both RDFox and TriAD delivered query answers to the coordinator, but did not store or print them. We evaluated each query five times, and in each run we recorded the query evaluation times as reported by each system, the network communication as reported by running `iptraf` on each server, and the total maximum memory use across all servers. We did not include the query planning times in the query response times. TriAD uses an in-built query planner based on graph summarisation, whereas RDFox uses an external prototypical implementation of the query planner described in Chapter 6. RDFox servers used five processing and one communication thread, and TriAD determined the number of threads internally. We used a cluster of ten m4.2xlarge Amazon EC2 servers connected by a network with a 1 Gb/s bandwidth, with each server having 32 GB RAM and eight virtual cores of 2.4 GHz Intel Xeon E5-2676v3 CPUs.

We have also included results from the literature for two more recent system: S2RDF [57] and PEDA [50]. The former is based on Apache Spark and was shown

to outperform H2RDF+ [49], Sempala [56], PigSPARQL [55], and SHARD [53]; the latter was shown to be comparable in performance to EAGRE [75], GraphPartition [32], SHAPE [43], FedX [59], and SPLENDID [23]. We did not repeat the relevant tests as both systems were tested on the same datasets and similar hardware,[2] and the reported results clearly show that S2RDF/PEDA are orders of magnitude slower than RDFox/TriAD on all but the IL-3-n queries.

8.2.2 Results

Tables 8.3 and 8.4 show the number of answers for each query, as well as the query times for WatDiv and LUBM respectively, while Tables 8.6 and 8.5 show the total network communication (i.e., the total amount of data sent by all servers), and the total memory (i.e., the sum of the maximum memory use of all servers). TriAD ran out of memory on eight queries marked '—', and S2RDF and PEDA were not tested on queries marked 'n/a'.

As one can see, all systems outperformed PEDA, which we believe to be due to PEDA's query evaluation strategy. The system first partially evaluates the given query and its subqueries on each server. This produces many complete answers, as well as partial answers that are combined in a separate assembly phase. This idea is closely related to our earlier work [51], where we observed that the number of partial answers can be prohibitive, particularly on large and complex queries. Indeed, even the partial query evaluation phase of PEDA (see [50]) is already slower by orders of magnitude than full query evaluation in RDFox or TriAD.

Apart from queries IL-3-n, RDFox and TriAD outperformed S2RDF by up to four orders of magnitude. To understand why, note that Spark partitions data at the level of blocks of a distributed file system, rather than at the level of triples. Triples are thus assigned to servers randomly, and not even triples with the same

[2]Both systems were tested on clusters of ten servers, but S2RDF used 1.9 GHz processors, and PEDA used 16 GB of RAM per server.

Table 8.3: Response times over WatDiv-10K test dataset

			Query Evaluation Time (ms)				
	Query	Answers	RDFox$_{GP}$	RDFox$_{HP}$	TriAD	S2RDF [57]	PEDA [50]
WatDiv-10K basic queries	L1	2	2	2	11	471	15,356
	L2	16,132	45	38	15	498	1,622
	L3	24	2	2	6	549	16,889
	L4	5,782	16	4	5	209	261
	L5	12,957	23	28	17	270	49,539
	S1	12	3	5	41	2,208	43,803
	S2	6,685	11	12	33	607	74,479
	S3	0	26	9	8	311	8,087
	S4	153	21	23	22	329	16,520
	S5	0	12	8	—	260	1,861
	S6	453	8	4	8	235	50,865
	S7	0	2	2	3	420	56,784
	F1	324	42	15	15	590	64,748
	F2	188	8	5	263	1,226	207,725
	F3	865	12	6	208	1,969	4,831,257
	F4	2,879	25	11	—	1,265	260,410
	F5	65	3	4	348	2,254	26,208
	C1	1,504	154	201	248	2,508	212,129
	C2	288	175	198	343	2,740	1,787,692
	C3	42.44 M	270	295	419	16,407	123,349
WatDiv-10K incremental linear queries	IL-1-5	7,469	3	4	8,322	12,543	n/a
	IL-1-6	0	3	4	19,957	12,252	n/a
	IL-1-7	16,132	32	35	32,496	15,062	n/a
	IL-1-8	0	3	5	17,985	15,003	n/a
	IL-1-9	1,591	24	5	8,505	15,478	n/a
	IL-1-10	758	12	19	8,294	16,124	n/a
	IL-2-5	15,444	4	5	8,241	41,188	n/a
	IL-2-6	0	3	3	14,471	13,276	n/a
	IL-2-7	1,386	9	12	3,600	14,182	n/a
	IL-2-8	267	6	7	18,128	15,261	n/a
	IL-2-9	171	6	7	18,021	16,313	n/a
	IL-2-10	32	8	10	17,718	13,922	n/a
	IL-3-5	3.34 G	773,619	782,884	—	29,590	n/a
	IL-3-6	3.75 G	1,770,593	2,042,988	—	87,525	n/a
	IL-3-7	584.92 M	1,239,115	1,307,131	—	102,971	n/a
	IL-3-8	19.56 G	6,996,328	7,379,970	—	2,068,100	n/a
	IL-3-9	954.41 M	2,254,083	2,452,539	—	158,595	n/a
	IL-3-10	954.41 M	2,090,013	2,154,819	—	141,940	n/a

Table 8.4: Response times over LUBM-10K test dataset

			Query Evaluation Time (ms)				
	Query	Answers	RDFox$_{GP}$	RDFox$_{HP}$	TriAD	S2RDF [57]	PEDA [50]
LUBM-10K	T1	2,528	727	1,017	13,410	n/a	309,361
	T2	10.80 M	699	975	927	n/a	23,685
	T3	0	419	1,149	771	n/a	10,368
	T4	10	4	4	7	n/a	753
	T5	10	2	2	2	n/a	125
	T6	125	3	3	85	n/a	1,914
	T7	439,994	1,010	14,574	7,294	n/a	46,123
	N1	2,528	1,723	9,263	1,755	n/a	n/a
	N2	4.11 M	4,487	32,226	23,711	n/a	n/a
	N3	2.22 M	920	6,297	33,661	n/a	n/a

Table 8.5: Network communication and RAM use over LUBM-10K test dataset

		Total Network Communication (kB)			Total RAM Use (MB)		
	Query	RDFox$_{GP}$	RDFox$_{HP}$	TriAD	RDFox$_{GP}$	RDFox$_{HP}$	TriAD
LUBM-10K	T1	4,033	716,588	197,762	37	148	1,144
	T2	150,073	151,466	104,657	144	153	154
	T3	1,897	1,915	466	1	12	708
	T4	51	44	115	1	8	1
	T5	18	17	63	1	1	1
	T6	38	41	153	1	1	1
	T7	10,857	1,454,774	29,592	8	90	844
	N1	5,487	1,742,285	8,154	21	48	232
	N2	138,803	4,973,168	184,661	72	280	3,501
	N3	65,539	1,204,533	111,571	24	4	6,645

subject are guaranteed to be colocated; hence, every join in a query plan requires data exchange, even for subject–subject joins. Moreover, servers in Spark are synchronised (i.e., all servers process the same part of the query plan at any point in time), and so there is less potential for parallelism compared to the asynchronous approaches used in RDFox and TriAD.

On the IL-3-n queries, S2RDF is up to an order of magnitude faster than the two RDFox variants, and TriAD ran out of memory in all cases. These queries were designed as a 'torture test' for RDF stores, and, as Table 8.3 shows, they

Table 8.6: Network communication and RAM use over WatDiv-10K test dataset

		Total Network Communication (kB)			Total RAM Use (MB)		
	Query	RDFox$_{GP}$	RDFox$_{HP}$	TriAD	RDFox$_{GP}$	RDFox$_{HP}$	TriAD
WatDiv-10K basic queries	L1	31	30	227	1	2	1
	L2	262	257	1,106	1	1	1
	L3	17	17	76	1	1	1
	L4	93	98	299	1	1	1
	L5	304	300	940	1	1	1
	S1	77	81	142	1	1	1
	S2	184	180	517	1	1	1
	S3	37	37	91	1	1	1
	S4	3,061	3,010	108	1	2	1
	S5	37	38	—	1	2	—
	S6	37	38	151	1	1	1
	S7	28	29	58	1	1	1
	F1	855	1,693	265	1	2	1
	F2	109	102	11,461	1	2	25
	F3	197	269	337	1	1	29
	F4	437	399	—	1	1	—
	F5	59	62	29,900	1	1	76
	C1	6,103	17,209	3,170	16	5	27
	C2	8,564	12,772	45,520	2	14	97
	C3	1,186	1,175	423	2	72	8
WatDiv-10K incremental linear queries	IL-1-5	285	284	2,276,527	800	881	18,492
	IL-1-6	54	55	6,595,267	269	217	37,599
	IL-1-7	1,624	2,302	9,370,381	138	1,186	49,780
	IL-1-8	69	71	5,756,890	33	36	28,713
	IL-1-9	1,972	2,106	3,061,088	21	28	18,605
	IL-1-10	1,032	979	3,097,132	23	22	18,604
	IL-2-5	505	518	2,970,367	4	9	18,430
	IL-2-6	56	55	5,149,633	1	5	27,260
	IL-2-7	1,428	796	358,506	2	1	7,430
	IL-2-8	519	282	5,302,047	5	1	28,660
	IL-2-9	336	257	5,457,976	1	3	28,896
	IL-2-10	768	565	5,131,115	3	1	28,812
	IL-3-5	105,472,000	112,019,456	—	2,794	2,844	—
	IL-3-6	263,450,624	284,635,136	—	6,583	5,507	—
	IL-3-7	72,719,360	99,340,288	—	5,264	2,988	—
	IL-3-8	307,225,600	1,007,030,272	—	6,805	7,314	—
	IL-3-9	153,052,160	185,125,888	—	9,276	1,803	—
	IL-3-10	175,571,968	208,282,624	—	4,881	2,853	—

return several orders of magnitude more answers than the other queries. These results are explained by the fact that query evaluation in Spark materialises the answers in a distributed file system. In contrast, our approach sends all answers to the coordinator, which becomes a major source of overhead for such queries. For example, during the evaluation of IL-3-5 on RDFox$_{GP}$, about 96% of all network traffic (about 107 GB) is used for sending answers, which would require 856 s using the full bandwidth of Gigabit Ethernet. In practice, RDFox$_{GP}$ evaluates the query in 773 s, which is possible since not all data is sent from the same server. The overhead of sending answers varies across queries (e.g., about 56% of network traffic is used on IL-3-6), but it is substantial in each case. This skews the comparison between RDFox and S2RDF on queries with large answer sets. Finally, if the user does not wish to iterate over all answers, our approach can easily be modified to store answers on the server where they are produced.

The performance of TriAD and RDFox is comparable on the WatDiv linear (Ln) and star (Sn) queries. Although TriAD uses a summary graph [28] to prune the search space, RDFox variants outperformed TriAD on the snowflake (Fn) and complex (Cn) queries, sometimes by up to two orders of magnitude (e.g., on F2, F3, and F5). Moreover, RDFox variants used less network communication on all queries except S4, F1, C1 and C3, but they were nevertheless quicker on S4, C1, and C3; hence, network use is not always a bottleneck, which we take into account during query planning.

Both RDFox variants outperformed TriAD, by between one and four orders of magnitude, on the extended WatDiv queries and on all LUBM queries apart from T5. In most cases, TriAD uses up to five orders of magnitude more network communication, and up to three orders of magnitude more memory. As already mentioned, TriAD evaluates its queries using bushy hash joins, whose memory use depends on the number of intermediate answers, and this prevented TriAD from processing any of the IL-3-n queries. In contrast, the memory use of index

Table 8.7: Idle memory use (without dictionaries) per server (GB)

Dataset	RDFox graph part.			RDFox hash part.			TriAD		
	Mean	Max	Sdev	Mean	Max	Sdev	Mean	Max	Sdev
WatDiv-10K	4.39	5.42	0.54	4.71	4.72	0.01	9.57	10.99	0.73
LUBM-10K	5.49	5.61	0.15	5.86	5.89	0.06	12.04	19.26	3.98

nested loop joins depends only on the number of query atoms; coupled with dynamic data exchange and careful flow control, this provides a powerful and scalable query evaluation technique.

RDFox$_{HP}$ was often slightly faster than RDFox$_{GP}$ on the WatDiv linear and star queries, and their performance was broadly the same on the IL-m-n queries since sending answers is the main source of overhead there. In contrast, RDFox$_{GP}$ performed slightly better than RDFox$_{HP}$ on all measures (time, communication, and memory use) on the snowflake and complex WatDiv queries and on LUBM. The uniform structure of LUBM is particularly amenable to graph partitioning, and the benefits of graph partitioning seem most significant there. In general, communication overhead seems to be lowest on RDFox$_{GP}$, followed by TriAD and then RDFox$_{HP}$. However, RDFox$_{HP}$ is still considerably faster than TriAD, and it uses much less memory on query T1.

Although RDFox$_{GP}$ does not seem to produce significant improvements in performance, RDFox$_{HP}$ outperforms the competing systems in most queries. This suggests that the effectiveness of dynamic data exchange is not largely impacted by the data partitioning strategy chosen, making it a flexible approach not relying on potentially expensive partitioning strategies such as graph partitioning.

Table 8.7 shows the average and maximal memory use per server after loading the data (excluding dictionaries), and the standard deviation across servers. Since RDFox does not duplicate data, it uses about half the memory of TriAD, which hashes its groups by subject and object. Also, memory use per server is more balanced for RDFox$_{GP}$ than for TriAD due to weighted graph partitioning.

Table 8.8: Minimum and maximum numbers of triples and average number of resources per partition element (in millions)

	WatDiv-10K			LUBM-10K		
Partitioning Scheme	Min	Max	Res	Min	Max	Res
Weighted, pruning	103.1	113.0	20.9	126.4	138.2	32.9
Weighted, no pruning	102.1	113.0	21.6	123.6	139.8	35.7
Unweighted, no pruning	22.5	410.7	18.1	123.7	142.3	36.0
Subject hashing	109.0	109.3	24.2	133.3	133.7	52.5

The experiments seem to suggest that network communication is not the only significant contributor to response times in distributed query evaluation. For example, query F5 shows TriAD having 3 orders of magnitude higher communication levels, yet it its response time was only 2 orders of magnitude higher. This suggests other aspects, such as query ordering and which physical join operators are used still play an important role, as in the centralised case.

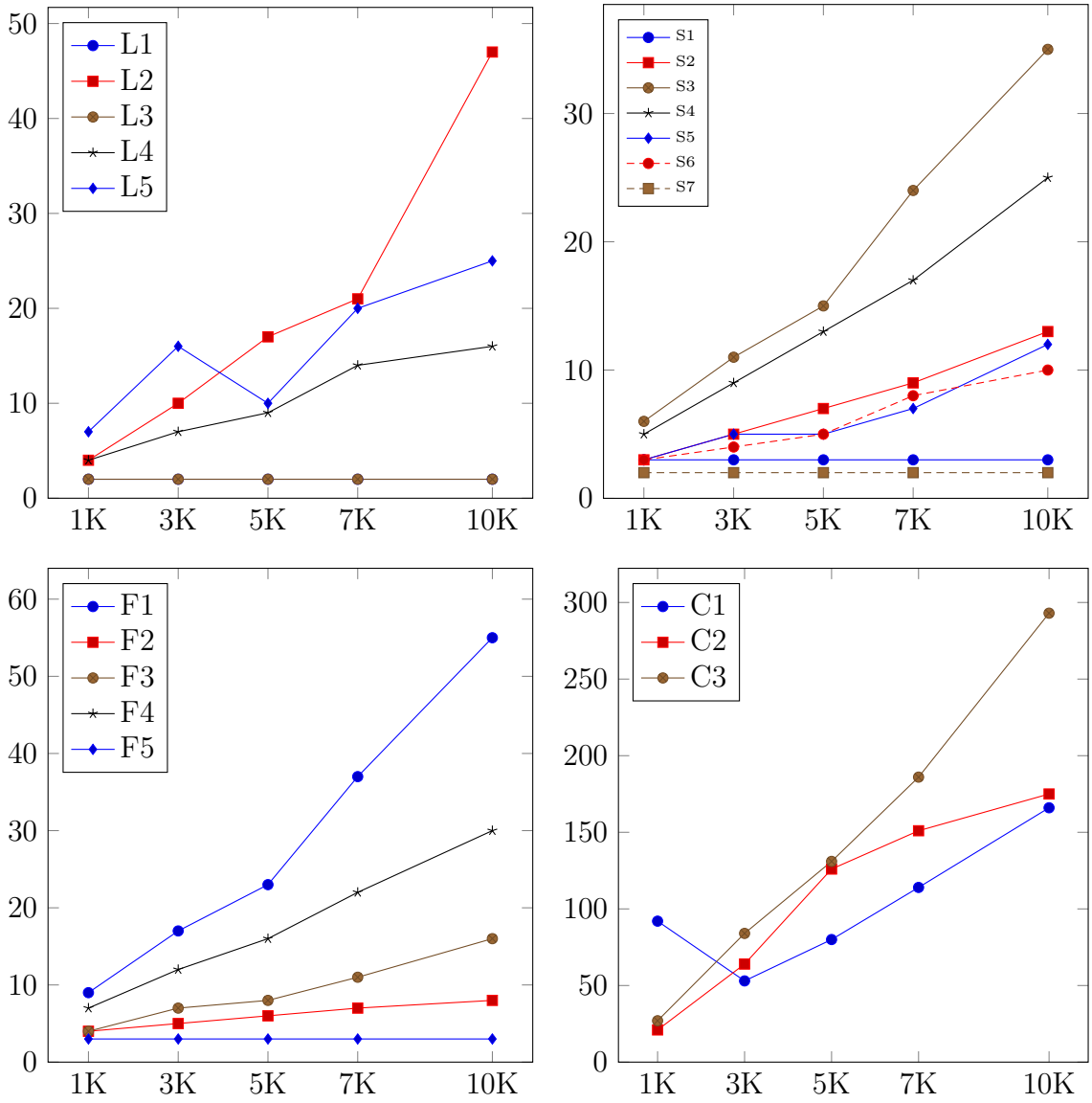
8.3 Data Partitioning Experiments

Table 8.8 shows the triple and resource distributions for the data partitioning strategy described in Chapter 7 (weighted, pruning), the strategy used without the pruning step (weighted, no pruning), vertex graph partitioning (unweighted, no pruning), and subject hashing. All of these approaches group triples by subject. As expected, subject hashing produces very balanced partitions. Moreover, weighted partitions are significantly more uniform than unweighted ones, particularly on WatDiv. The average number of resources per partition element is much smaller in partitioning-based approaches than for subject hashing, suggesting that connections are much better clustered in the former approaches.

8.4 Scalability Experiments

Figure 8.1 shows the scalability results for RDFox$_{GP}$. Five datasets were used, according to different scale factors of the WatDiv benchmark, with the number of

Figure 8.1: Scalability of RDFox over WatDiv (Dataset size by response time (ms))



triples in each dataset shown in Table 8.1. The cluster remained fixed, as in the previous experiments. The overall trend was consistent across most queries, with response time increasing linearly with the size of the data. Queries L1, L3, S1, S7, and F5 had roughly constant response times across all datasets as those queries all had a roughly constant number of answers, compared to the remaining queries whose answer count scaled roughly linearly with the data size.

8.5 Conclusion

Overall, our evaluation has shown promising results. We conducted an in-depth comparison of RDFox against state-of-the-art systems and showed that our approach outperforms these systems in response time, memory use, and network use, by several orders of magnitude in many cases. We were able to experimentally verify the correctness of our approach, particularly in terms of memory use: RDFox was able to answer every query correctly whereas the other systems in some cases ran out of memory and failed. Interestingly, the results showed that network communication is not always a significant factor, and those affecting centralised systems such as query planning, are still significant in the distributed setting. Furthermore, we showed that dynamic data exchange is not significantly affected by the partitioning approach. We showed that a computationally expensive, tailor made partitioning strategy (RDFox_{GP}) produced similar results to a simple and computationally cheap approach (RDFox_{HP}). This is promising as it extends the flexibility of the approach.

9 Conclusion & Outlook

In this thesis we presented many novel advancements to query answering in distributed RDF databases. We identified shortfalls of existing systems, such as missed local computation, redundant network communication and large storage overheads. To address such issues, we presented two query evaluation strategies: the first one is based on partial evaluation, and the second one is based on dynamic data exchange.

Our partial evaluation strategy introduced the wildcard resource which represented resources that were external to each server, and acted as an extreme summarisation of the rest of the graph. This resource was used to identify which answers spanned multiple partitions, effectively pruning partial answers which could not be extended on other servers, as well as aid in joining partial answers at the assembly stage. This approach meant the data partitioning scheme did not need to provide guarantees on data placement, as is common in existing systems, which gave the flexibility to adapt graph partitioning software to be used on RDF graphs, which we showed produced better quality partitions than similar systems, without a significant storage overhead. Although the partitioning quality meant many answers to common queries were local in our tests, the number of partial

answers produced were prohibitive in some cases. This led to the development of our second evaluation strategy.

One of the root causes of many inefficiencies that we identified in existing systems was the use of the data exchange operator, or some derivative. To solve this, we presented our novel dynamic data exchange approach, which shifted the decision of when and where to move data from compile time, as with the data exchange operator, to run time. This produced many benefits; our query answering algorithm guarantees that each local answer is computed locally, which in turn reduces the amount of network communication. Coupling this with index nested-loop evaluation and careful flow control, we prove strong memory bounds for evaluation, being linear in the number of atoms in the query. This guaranteed our solution would not run out of memory on complex queries with large numbers of intermediate answers, which we verified through experimental evaluation. As our algorithm was asynchronous, detecting termination was a non-trivial problem, so we presented a novel, lightweight termination detection algorithm. Furthermore, we presented optimisations to improve efficiency and reduce the amount of network communication.

The effectiveness of any query evaluation strategy is heavily dependent on query optimisation, as the amount of work done can differ by many orders of magnitude between good and bad query plans, which amount to reorderings of the atoms of the query in our framework. As there is a substantial body of work surrounding query optimisation, we presented methods of how to adapt existing techniques to accurately model our query evaluation strategy. This involved presenting methods for calculating the number of intermediate answers produced as well as the number of partial answers sent between servers. Moreover, we described an algorithm to compute the lowest cost plan while taking into account the parallel nature of distributed evaluation.

Similarly to our partial evaluation strategy, dynamic data exchange does not rely on guarantees provided by the data partitioning scheme. We took advantage of

this flexibility, building on the data partitioning scheme from our partial evaluation strategy to present a novel data partitioning scheme based on weighted graph partitioning. The use of weighted graph partitioning applied to RDF graphs meant we were able to produce partitions that were balanced in the number of triples rather than resources. We showed that the former was much robust than the latter, which in some cases can produce extremely unbalanced partitions. Importantly, we also showed that the performance of our query evaluation strategy was not dependent on potentially computationally expensive strategies as the above, and in fact simple partitioning schemes like hash partitioning still provide exceptional and comparable performance.

Finally, we evaluated our approach experimentally, providing an in-depth comparison to state-of-the-art systems across two common benchmarks. We showed that our approach outperforms these systems in terms of response time, memory use, and network use, often by orders of magnitude.

The importance of our approach was further validated through collaboration with Oracle, who implemented it in their distributed graph database, PGX.D Async [54]. Our techniques were easily adapted to the property graph model, as opposed to RDF, and their implementation showed promising performance.

9.1 Future Work

We anticipate several directions for future work. We have provided the base framework for many problems within distributed processing of RDF data with our dynamic data exchange approach. The query answering algorithm in Chapter 5 only covers basic graph patterns, however, we believe it could be extended to cover all of SPARQL. For example, the UNION construct would be a trivial extension achieved by issuing multiple queries and combining the result. We expect property paths, introduced in SPARQL 1.1 [26], to be the most challenging as the length of

the answer is not necessarily predetermined. Additionally, OPTIONAL could prove challenging, as although outer joins can be easily implemented with nested loop joins, since we possibly fork execution across servers, ensuring the correct multiplicity is reported when the optional clause does not have a match is non-trivial.

In Chapter 5, we described the extent to which our query answering algorithm takes advantage of parallelism. This is achieved through multiple threads servicing the incoming message buffer and extending partial answers. However, each message is processed in a sequential manner by a single thread. As a result, the amount of parallelism is dependent on the distribution of messages and parallelism is limited if there are few or infrequent messages. There are many existing techniques for parallel query evaluation [8, 24, 70] which we expect would work efficiently in our framework.

Our techniques were implemented in RDFox [45], which offers efficient, in-memory materialisation of datalog programs over RDF data. At the moment, this is done in a centralised manner; however, we believe our approach to query answering can be extended to materialisation. The problems of query answering and materialisation are very related, and indeed the materialisation strategy used in RDFox issues queries as a primary mechanism. One of the key elements to efficiency in materialisation is avoiding repeated derivations, which is done in RDFox in part by maintaining an order of the triples. It is more challenging to define an order over distributed data, so we anticipate avoiding repeated derivations to be significantly more challenging in the distributed setting.

Non-uniform memory access (NUMA) architectures pose numerous performance challenges for information systems, and have seen increased prevalence in modern systems. There are many parallels between NUMA and distributed data access. For this reason, we believe our approach could be adapted for use in high-performance NUMA databases.

Appendices

A

Queries Used in the Evaluation

A.1 WatDiv Queries

These queries were used for the WatDiv benchmark evaluation. For each query, the atoms are ordered as produced by the query planner. The prefixes used are below. Variables surrounded in % characters, such as $\%v1\%$ are placeholders that get instantiated with constants by the query generator.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wsdbm: <http://db.uwaterloo.ca/~galuc/wsdbm/>
PREFIX sorg: <http://schema.org/>
PREFIX gn: <http://www.geonames.org/ontology#>
PREFIX og: <http://ogp.me/ns#>
PREFIX gr: <http://purl.org/goodrelations/>
PREFIX dc: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/>
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX rev: <http://purl.org/stuff/rev#>
```

```
L1.
SELECT ?v0 ?v2 ?v3 WHERE {
  ?v0    wsdbm:subscribes    %v1% .
  ?v0    wsdbm:likes    ?v2 .
  ?v2    sorg:caption    ?v3 .
}
```

```
L2.
SELECT ?v1 ?v2 WHERE {
  %v0%    gn:parentCountry    ?v1 .
  ?v2    sorg: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    og:tag    %v1% .
  ?v0    sorg:caption    ?v2 .
}
```

```
L5.
SELECT ?v0 ?v1 ?v3 WHERE {
  %v2%    gn:parentCountry    ?v3 .
  ?v0    sorg:nationality    ?v3 .
  ?v0    sorg:jobTitle    ?v1 .
}


S1.
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9 WHERE {
  %v2%    gr:offers    ?v0 .
  ?v0    sorg:priceValidUntil    ?v9 .
  ?v0    gr:validFrom    ?v5 .
  ?v0    gr:validThrough    ?v6 .
  ?v0    gr:serialNumber    ?v4 .
  ?v0    sorg:eligibleQuantity    ?v7 .
  ?v0    sorg:eligibleRegion    ?v8 .
  ?v0    gr:includes    ?v1 .
  ?v0    gr:price    ?v3 .
}


S2.
SELECT ?v0 ?v1 ?v3 WHERE {
  ?v0    sorg:nationality    %v2% .
  ?v0    rdf:type    wsdbm:Role2 .
  ?v0    dc:Location    ?v1 .
  ?v0    wsdbm:gender    ?v3 .
}


S3.
SELECT ?v0 ?v2 ?v3 ?v4 WHERE {
  ?v0    sorg:publisher    ?v4 .
  ?v0    rdf:type    %v1% .
  ?v0    sorg:caption    ?v2 .
  ?v0    wsdbm:hasGenre    ?v3 .
}


S4.
SELECT ?v0 ?v2 ?v3 WHERE {
  ?v0    sorg:nationality    wsdbm:Country1 .
  ?v0    foaf:age    %v1% .
  ?v0    foaf:familyName    ?v2 .
  ?v3    mo:artist    ?v0 .
}


S5.
SELECT ?v0 ?v2 ?v3 WHERE {
  ?v0    sorg:language    wsdbm:Language0 .
  ?v0    rdf:type    %v1% .
  ?v0    sorg:keywords    ?v3 .
  ?v0    sorg:description    ?v2 .
}


S6.
SELECT ?v0 ?v1 ?v2 WHERE {
  ?v0    wsdbm:hasGenre    %v3% .
  ?v0    mo:conductor    ?v1 .
  ?v0    rdf:type    ?v2 .
}


S7.
SELECT ?v0 ?v1 ?v2 WHERE {
  %v3%    wsdbm:likes    ?v0 .
  ?v0    sorg:text    ?v2 .
  ?v0    rdf:type    ?v1 .
}
```

```
F1.
SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE {
  ?v3    rdf:type    wsdbm:ProductCategory2 .
  ?v3    sorg:trailer    ?v4 .
  ?v3    wsdbm:hasGenre    ?v0 .
  ?v0    og:tag    %v1% .
  ?v0    rdf:type    ?v2 .
  ?v3    sorg:keywords    ?v5 .
}


F2.
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 WHERE {
  ?v0    wsdbm:hasGenre    %v8% .
  ?v0    sorg:caption    ?v4 .
  ?v0    foaf:homepage    ?v1 .
  ?v1    sorg:url    ?v6 .
  ?v1    wsdbm:hits    ?v7 .
  ?v0    sorg:description    ?v5 .
  ?v0    rdf:type    ?v3 .
  ?v0    og:title    ?v2 .
}


F3.
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 WHERE {
  ?v0    wsdbm:hasGenre    %v3% .
  ?v0    sorg:contentSize    ?v2 .
  ?v0    sorg:contentRating    ?v1 .
  ?v5    wsdbm:purchaseFor    ?v0 .
  ?v5    wsdbm:purchaseDate    ?v6 .
  ?v4    wsdbm:makesPurchase    ?v5 .
}


F4.
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
  ?v0    og:tag    %v3% .
  ?v0    foaf:homepage    ?v1 .
  ?v0    sorg:description    ?v4 .
  ?v0    sorg:contentSize    ?v8 .
  ?v1    sorg:language    wsdbm:Language0 .
  ?v1    sorg:url    ?v5 .
  ?v1    wsdbm:hits    ?v6 .
  ?v2    gr:includes    ?v0 .
  ?v7    wsdbm:likes    ?v0 .
}


F5.
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
  %v2%    gr:offers    ?v0 .
  ?v0    gr:includes    ?v1 .
  ?v0    gr:price    ?v3 .
  ?v0    gr:validThrough    ?v4 .
  ?v1    og:title    ?v5 .
  ?v1    rdf:type    ?v6 .
}


C1.
SELECT ?v0 ?v4 ?v6 ?v7 WHERE {
  ?v0    sorg:caption    ?v1 .
  ?v0    sorg:contentRating    ?v3 .
  ?v0    sorg:text    ?v2 .
  ?v0    rev:hasReview    ?v4 .
  ?v4    rev:title    ?v5 .
  ?v4    rev:reviewer    ?v6 .
  ?v7    sorg:actor    ?v6 .
  ?v7    sorg:language    ?v8 .
}
```

```
C2.
SELECT ?v0 ?v3 ?v4 ?v8 WHERE {
  ?v4    sorg:jobTitle    ?v5 .
  ?v4    foaf:homepage    ?v6 .
  ?v4    wsdbm:makesPurchase    ?v7 .
  ?v7    wsdbm:purchaseFor    ?v3 .
  ?v2    gr:includes    ?v3 .
  ?v2    sorg:eligibleRegion    wsdbm:Country5 .
  ?v3    rev:hasReview    ?v8 .
  ?v8    rev:totalVotes    ?v9 .
  ?v0    gr:offers    ?v2 .
  ?v0    sorg:legalName    ?v1 .
}

C3.
SELECT ?v0 WHERE {
  ?v0    wsdbm:likes    ?v1 .
  ?v0    dc:Location    ?v3 .
  ?v0    foaf:age    ?v4 .
  ?v0    wsdbm:gender    ?v5 .
  ?v0    foaf:givenName    ?v6 .
  ?v0    wsdbm:friendOf    ?v2 .
}
```

A.2 LUBM Queries

These queries were used for the LUBM benchmark evaluation. For each query, the atoms are ordered as produced by the query planner. The prefixes used are below.

```
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

T1.
SELECT ?X ?Y ?Z WHERE {
  ?Y rdf:type ub:University .
  ?X ub:undergraduateDegreeFrom ?Y .
  ?X ub:memberOf ?Z .
  ?Z ub:subOrganizationOf ?Y .
  ?Z rdf:type ub:Department .
  ?X rdf:type ub:GraduateStudent .
}

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 ub:worksFor <http://www.Department0.University0.edu> .
   ?X rdf:type ub:FullProfessor .
   ?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 ?P1 ?D1 ?S1 ?U1 WHERE {
  ?D1 ub:subOrganizationOf ?U1 .
  ?P1 ub:worksFor ?D1 .
  ?S1 ub:advisor ?P1 .
  ?S1 ub:undergraduateDegreeFrom ?U1 .
}

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

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

Bibliography

[1] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. "Scalable semantic web data management using vertical partitioning". In: *Proceedings of the VLDB Endowment*. VLDB Endowment. 2007, pp. 411–422.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] Ashraf Aboulnaga and Surajit Chaudhuri. "Self-tuning Histograms: Building Histograms Without Looking at Data". In: *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. 1999, pp. 181–192.

[4] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. "Diversified Stress Testing of RDF Data Management Systems". In: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*. 2014, pp. 197–212.

[5] Konstantin Andreev and Harald Racke. "Balanced graph partitioning". In: *Theory of Computing Systems* 39.6 (2006), pp. 929–939.

[6] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. "Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data". In: *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*. 2010, pp. 41–50.

[7] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. "Dbpedia: A nucleus for a web of open data". In: *The Semantic Web Journal* (2007), pp. 722–735.

[8] Paul Beame, Paraschos Koutris, and Dan Suciu. "Communication Steps for Parallel Query Processing". In: *Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '13. New York, NY, USA: ACM, 2013, pp. 273–284.

[9] Dan Brickley and R. V. Guha. *RDF Schema 1.1*. 2014.

[10] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. "STHoles: A Multidimensional Workload-Aware Histogram". In: (2001), pp. 211–222.

[11] Peter Buneman, Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. "Using Partial Evaluation in Distributed Query Evaluation". In: *Proceedings of the VLDB Endowment*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 211–222.

[12] Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. "Distributed Query Evaluation with Performance Guarantees". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 509–520.

[13] Gao Cong, Wenfei Fan, Anastasios Kementsietsidis, Jianzhong Li, and Xianmin Liu. "Partial Evaluation for Distributed XPath Query Processing and Beyond". In: *ACM Transactions on Database Systems* 37.4 (Dec. 2012), 32:1–32:43.

[14] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[15] Chris HQ Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and Horst D Simon. "A min-max cut algorithm for graph partitioning and data clustering". In: *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE. 2001, pp. 107–114.

[16] Wenfei Fan, Xin Wang, and Yinghui Wu. "Performance Guarantees for Distributed Reachability Queries". In: *Proceedings of the VLDB Endowment* 5.11 (2012), pp. 1304–1315.

[17] Wenfei Fan, Xin Wang, Yinghui Wu, and Dong Deng. "Distributed Graph Simulation: Impossibility and Possibility". In: *Proceedings of the VLDB Endowment* 7.12 (Aug. 2014), pp. 1083–1094.

[18] David C Faye, Olivier Curé, 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.

[19] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994.

[20] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. "An Empirical Study of Real-World SPARQL Queries". In: *1st International Workshop on Usage Analysis and the Web of Data*. 2011.

[21] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. "Query Optimization for Parallel Execution". In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*. 1992, pp. 9–18.

[22] Lise Getoor, Benjamin Taskar, and Daphne Koller. "Selectivity Estimation using Probabilistic Models". In: (2001), pp. 461–472.

[23] Olaf Görlitz and Steffen Staab. "Splendid: SPARQL endpoint federation exploiting void descriptions". In: *2nd International Conference on Consuming Linked Data*. Vol. 782. CEUR Workshop Proceedings. 2011, pp. 13–24.

[24] Goetz Graefe. "Query evaluation techniques for large databases". In: *ACM Computing Surveys* 25.2 (1993), pp. 73–169.

[25] Goetz Graefe and Diane L. Davison. "Encapsulation of parallelism and architecture-independence in extensible database query execution". In: *IEEE Transactions on Software Engineering* 19.8 (1993), pp. 749–764.

[26] The W3C SPARQL Working Group. *SPARQL 1.1 Overview*. 2013.

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

[28] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. "TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. 2014, pp. 289–300.

[29] 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: *Proceedings of the VLDB Endowment* 8.6 (2015), pp. 654–665.

[30] Steve Harris, Nick Lamb, and Nigel Shadbolt. "4store: The design and implementation of a clustered RDF store". In: *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*. 2009.

[31] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. "YARS2: A Federated Repository for Querying Graph Structured Data from the Web". In: *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*. 2007, pp. 211–224.

[32] Jiewen Huang, Daniel J Abadi, and Kun Ren. "Scalable SPARQL Querying of Large RDF Graphs". In: *Proceedings of the VLDB Endowment* 4.11 (2011), pp. 1123–1134.

[33] Mohammad Husain, James McGlothlin, Mohammad M Masud, Latifur Khan, and Bhavani M Thuraisingham. "Heuristics-based query processing for large

RDF graphs using cloud computing". In: *IEEE Transactions on Knowledge and Data Engineering* 23.9 (2011), pp. 1312–1327.

[34] Mohammad Farhan Husain, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. "Data intensive query processing for large RDF graphs using cloud computing tools". In: *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE. 2010, pp. 1–10.

[35] Cray Inc. *Cray® Urika®-GX Agile Analytics Platform for Financial Services*. 2016. URL: `https://www.cray.com/sites/default/files/Urika-GX-Financial-Services.pdf`.

[36] Neil D. Jones. *Challenging Problems in Partial Evaluation and Mixed Computation*. Vol. 6. 2&3. 1988, pp. 291–302.

[37] Neil D. Jones. "An Introduction to Partial Evaluation". In: *ACM Computing Surveys* 28.3 (Sept. 1996), pp. 480–503.

[38] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

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

[40] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM Journal of Scientific Computing* 20.1 (1998), pp. 359–392.

[41] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. "Scan-sharing for optimizing rdf graph pattern matching on mapreduce". In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE. 2012, pp. 139–146.

[42] Graham Klyne, Jeremy J. Carroll, and Brian McBride. *RDF 1.1: Concepts and Abstract Syntax*. 2014.

[43] Kisung Lee and Ling Liu. "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning". In: *Proceedings of the VLDB Endowment* 6.14 (2013), pp. 1894–1905.

[44] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. "Distributed Graph Pattern Matching". In: *Proceedings of the 21st International Conference on World Wide Web*. WWW '12. New York, NY, USA: ACM, 2012, pp. 949–958.

[45] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. "RDFox: A Highly-Scalable RDF Store". In: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*. 2015, pp. 3–20.

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

[47] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1099–1110.

[48] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. "H2RDF: adaptive query processing on RDF data in the cloud". In: *Proceedings of the 21st World Wide Web Conference, WWW 2012, Lyon, France, April 16-20, 2012 (Companion Volume)*. 2012, pp. 397–400.

[49] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. "H2RDF+: High-performance distributed joins over large-scale RDF graphs". In: *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*. 2013, pp. 255–263.

[50] 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.

[51] Anthony Potter, Boris Motik, and Ian Horrocks. "Querying Distributed RDF Graphs: The Effects of Partitioning". In: *Proceedings of the 10th International Workshop on Scalable Semantic Web Knowledge Base Systems co-located with 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 20, 2014*. 2014, pp. 29–44.

[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] Kurt Rohloff and Richard E. Schantz. "Clause-iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-store". In: *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing*. DIDC '11. San Jose, California, USA: ACM, 2011, pp. 35–44.

[54] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. "PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine". In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. GRADES'17. Chicago, IL, USA: ACM, 2017, 7:1–7:6.

[55] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, and Georg Lausen. "PigSPARQL: A SPARQL Query Processing Baseline for Big Data". In: *Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013*. 2013, pp. 241–244.

[56] Alexander Schätzle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg Lausen. "Sempala: Interactive SPARQL Query Processing on Hadoop". In: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*. 2014, pp. 164–179.

[57] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. "S2RDF: RDF querying with SPARQL on spark". In: *Proceedings of the VLDB Endowment* 9.10 (2016), pp. 804–815.

[58] Michael Schmidt, Thomas Hornung, Michael Meier, Christoph Pinkel, and Georg Lausen. "SP^{2}Bench: A SPARQL Performance Benchmark". In: *Semantic Web Information Management - A Model-Based Perspective*. 2009, pp. 371–393.

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

[60] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. "Access path selection in a relational database management system". In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM. 1979, pp. 23–34.

[61] 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*. ACM. 2013, pp. 505–516.

[62] Lefteris Sidirourgos, Romulo Gonçalves, Martin Kersten, Niels Nes, and Stefan Manegold. "Column-Store Support for RDF Data Management: not all swans are white". In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1553–1563.

[63] Daniel A Spielman and Shang-Hua Teng. "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems". In: *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM. 2004, pp. 81–90.

[64] Isabelle Stanton and Gabriel Kliot. "Streaming graph partitioning for large distributed graphs". In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2012, pp. 1222–1230.

[65] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. "Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation". In: *CoRR* abs/1801.09619 (2018).

[66] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. "FENNEL: Streaming Graph Partitioning for Massive Scale Graphs". In: *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. WSDM '14. New York, New York, USA: ACM, 2014, pp. 333–342.

[67] Deepak Vohra. *Apache HBase Primer*. 1st. Berkely, CA, USA: Apress, 2016.

[68] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. "Hexastore: sextuple indexing for semantic web data management". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 1008–1019.

[69] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[70] Annita N. Wilschut and Peter M. G. Apers. "Dataflow query execution in a parallel main-memory environment". In: *Distributed and Parallel Databases* 1.1 (1993), pp. 103–128.

[71] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Hai Jin, and Ling Liu. "SemStore: A Semantic-Preserving Distributed RDF Triple Store". In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*. 2014, pp. 509–518.

[72] 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.

[73] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[74] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. "A Distributed Graph Engine for Web Scale RDF Data". In: *Proceedings of the VLDB Endowment* 6.4 (2013), pp. 265–276.

[75] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. "EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 2013, pp. 565–576.

[76] 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.