Distributed dataflow processing of large RDF graphs

Maali, Fadi

2017-05-29

http://hdl.handle.net/10379/6674
Distributed Dataflow Processing of Large RDF Graphs

by

Fadi Maali

Supervisor: Prof. Dr. Stefan Decker
A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
College of Engineering and Informatics
NUI Galway

July 2017
“Mathematical elegance and practical utility are close companions, as the history of science has shown again and again. Sometimes scientists discover the theory and leave it to mathematicians to figure out why it’s elegant, and other times mathematicians develop an elegant theory and leave it to scientists to figure out what it’s good for.”

How not to be Wrong, Jordan Ellenberg

“Mathematical solutions are selected by the subliminal self on the basis of mathematical beauty, of the harmony of numbers and forms, of geometric elegance.”

Zen and the Art of Motorcycle Maintenance, Robert M Pirsig
Abstract

As part of the big data world, RDF, the graph-based data model of the Semantic Web, is growing in use. Consequently, the size of available RDF data is increasing and massive datasets are becoming commonplace. Nevertheless, when analysing large RDF datasets, users are left mainly with two options: using SPARQL, the main query language for RDF, or using an existing non-RDF-specific big data language. This thesis argues that each of these two approaches has its own limitations. SPARQL is costly to compute and complex analyses can be hard to express in a purely declarative SPARQL query. On the other hand, using existing big data languages designed for tabular data commonly results in verbose, unreadable, and sometimes inefficient scripts. This dissertation, therefore, pursues defining a dataflow language specifically designed to process large RDF data on top of distributed platforms.

In developing a dataflow language, this dissertation discusses three components: (i) the data query language, (ii) the underlying data model, and (iii) the physical arrangement of the underlying distributed data.

On data models, we introduce $\text{RDF.co}$, a data model that defines a pair of a binding and a graph in the value of each expression. Compared to the SPARQL algebra, $\text{RDF.co}$ is fully composable. We provide a formal definition of the syntax and semantics of the data model, characterise its expressivity in comparison to SPARQL, and present a number of its unique algebraic properties. Algebraic properties of $\text{RDF.co}$ represent a unique study on relations between triple patterns. These properties, when interpreted as rewriting rules, provide theoretical foundation needed to apply cost-based query optimisation. We present rules for triple patterns elimination, insertion, and pushing down.

On physical arrangement of the underlying distributed data, this dissertation focuses on graph partitioning. We define an RDF graph partitioning approach using pattern matching. This use of pattern matching allows mapping query answering over partitioned graphs to the well-studied problem of view-based query answering. Our experiments show that using pattern matching to guide graph partitioning allows leveraging knowledge that might be available about the data or the task at hand to enhance query answering time.
On data query languages, this dissertation describes SYRql, a dataflow language for large scale processing of RDF data. We describe an implementation of SYRql on top of the MapReduce platform. SYRql implementation utilises the underlying data model and the partitioning of the data. Our experiments show that SYRql performance for query answering is comparable to that of other well-established big data languages such as IBM Jaql and the Apache Pig Latin.
Acknowledgements

Fadi Maali is funded by the Irish Research Council, Embark Postgraduate Scholarship Scheme
# Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xv</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Thesis Context ............................................. 2
   1.1.1 On Data Query Languages .............................. 3
   1.1.2 On Data Models ....................................... 4
   1.1.3 On Managing Graph Data ............................... 5
1.2 Thesis Approach and Hypotheses ............................ 6
   1.2.1 On Managing Graph Data ................................ 6
   1.2.2 On Data Models ....................................... 6
   1.2.3 On Data Query Languages .............................. 7
1.3 Contributions and Thesis Outline .......................... 7
   1.3.1 On Data Models ....................................... 7
   1.3.2 On Graph Partitioning ................................ 8
   1.3.3 On Data Query Languages .............................. 9
   1.3.4 Summary of Publications .............................. 9
1.4 Research Methodology ..................................... 10

## 2 Preliminaries

2.1 RDF & SPARQL ................................................ 11
   2.1.1 The SPARQL Algebra ................................... 12
   2.1.2 Composability of the SPARQL Algebra .................. 14
   2.1.3 Complexity of SPARQL Evaluation ...................... 14
2.2 Big Data Systems ........................................... 15
   2.2.1 The MapReduce Model .................................. 15
   2.2.2 Vertex-centric Model .................................. 16
2.3 Big Data Systems & RDF .................................... 17
2.4 Dataflow Languages Definition ............................. 18
2.5 Existing Data Languages ................................... 18
   2.5.1 Big Data Languages ................................. 19
2.5.2 RDF Data Languages ........................................... 20
2.5.3 Graph Languages .............................................. 21
2.5.4 NoSQL Systems ................................................ 21
2.5.5 Summary and Requirements ................................. 22
2.6 Summary .......................................................... 23

I RDF.co, the Data Model ........................................... 25

3 RDF.co, Composable Operators to Manipulate RDF Data ............ 27
  3.1 RDF.co Expressions ............................................ 28
     3.1.1 Preliminaries ............................................. 28
     3.1.2 Syntax .................................................... 28
     3.1.3 Semantics ............................................... 29
  3.2 Properties of RDF.co Expressions ............................. 35
  3.3 Derived Expressions .......................................... 39
     3.3.1 Graph Access (gAccess) ................................. 39
     3.3.2 Equi Join (E-JOIN) .................................... 40
     3.3.3 Graph Pattern Matching (GMATCH) ..................... 40
     3.3.4 Multiple Graph Extensions (mgEXT) .................. 42
  3.4 Relationship to SPARQL ....................................... 44
     3.4.1 RDF Datasets in SPARQL ............................... 44
     3.4.2 Expressing SPARQL SELECT Queries Using RDF.co Expressions ............................................. 45
     3.4.3 Expressing SPARQL CONSTRUCT Queries Using RDF.co Expressions ............................................. 46
     3.4.4 Discussion ................................................. 47
     3.4.4.1 The Set of Operators in RDF.co .................. 47
     3.4.4.2 Nesting Queries in SPARQL ......................... 48
     3.4.4.3 RDF.co Limitations ................................ 49
  3.5 Summary ........................................................ 49

4 Algebraic Properties of RDF.co Expressions ......................... 51
  4.1 Cascading Triple Pattern Matching Operators .................. 52
     4.1.1 Triple Pattern Elimination ............................. 55
     4.1.2 Triple Pattern Insertion ................................ 56
     4.1.3 Triple Pattern Push-down ............................... 58
  4.2 Evaluating Graph and Binding Components of an Expression ... 60
     4.2.1 Component-based Equivalence between RDF.co Expressions ............................................. 60
     4.2.2 Dependencies between Components of RDF.co Expressions Values ............................................. 61
  4.3 Summary ........................................................ 65

5 Extending RDF.co with Aggregation Operators ....................... 67
  5.1 Aggregation Operators (AGG) ................................ 67
  5.2 Graph Aggregation Operator (Gagg) .......................... 69
5.3 Properties of the Aggregation Operators
5.3.1 Length of Binding
5.3.2 Primitivity of the Aggregation Operators
5.3.3 Expressing SPARQL Aggregation
5.3.4 Cascading RDF.co Operators
5.3.5 Dependencies Between Graph and Binding Components
5.4 Summary

II Graph Partitioning, the Physical Data Layout

6 Pattern-based Graph Partitioning
6.1 Graph Partitioning and Distributed Platforms
6.2 Pattern Matching for Partitioning RDF Graphs
6.3 Related Work
6.3.1 Min-cut Partitioning
6.3.2 n-hops Partitioning
6.3.3 Defining Existing Partitioning Methods Using Graph Patterns
6.4 Summary

7 Partitioning RDF Graphs
7.1 Preliminaries
7.1.1 Graph Patterns Matching
7.2 Data Partitioning and Placement
7.2.1 Partitioning Scheme
7.2.2 Pattern-based Graph Partitioning
7.2.3 Data Placement
7.3 Query Answering
7.4 Related Work
7.5 Summary

8 Implementation and Experimental Study
8.1 Implementation
8.2 Experimental Study
8.2.1 Experiment Setup
8.2.2 Results and Discussion
8.3 Related Work
8.4 Limitations
8.5 Summary

III SYRql, the Language Layer

9 SYRql, a Dataflow Language for RDF
9.1 SYRql, a Dataflow Language for RDF
9.2 Execution and Optimisation of SYRql Programs .......................... 115
9.3 SYRql Implementation ......................................................... 116
  9.3.1 Data Representation .................................................. 117
  9.3.2 Parsing, Compiling and Execution ................................. 117
  9.3.3 Implemented Optimisations ........................................ 118
9.4 Evaluation ........................................................................... 119
  9.4.1 Experimental Setup ..................................................... 121
  9.4.2 Results & Discussion .................................................. 122
9.5 Related Work ...................................................................... 123
9.6 Summary ............................................................................ 124

10 Graph Aggregation in SPARQL ............................................ 125
  10.1 Motivation ........................................................................ 125
  10.2 Model .............................................................................. 126
    10.2.1 Preliminaries ............................................................. 127
    10.2.2 Graph Aggregation Operator ................................. 128
  10.3 Answering Graph Aggregation Queries ............................... 132
    10.3.1 Binding Table ............................................................ 132
    10.3.2 Building the Grouped Graph ............................... 133
    10.3.3 Applying the Graph Reduce Function .................... 133
  10.4 Evaluation ........................................................................ 134
    10.4.1 Expressivity ............................................................... 134
    10.4.2 Performance Evaluation ........................................ 135
      10.4.2.1 Type Summary ............................................... 136
      10.4.2.2 Bibliometrics ................................................... 136
    10.4.3 Discussion ................................................................. 137
  10.5 Related Work .................................................................. 139
  10.6 Summary ........................................................................ 139

11 Summary & Conclusions ....................................................... 141
  11.1 Findings ......................................................................... 144
    11.1.1 On Data Models ....................................................... 144
    11.1.2 On Graph Partitioning ............................................. 146
    11.1.3 On Data Query Languages .................................... 146
  11.2 Limitations ...................................................................... 147
    11.2.1 On Data Models ....................................................... 147
    11.2.2 On Graph Partitioning ............................................. 147
    11.2.3 On Data Query Languages .................................... 148
  11.3 Directions for Future Research ....................................... 148
    11.3.1 On Data Models ....................................................... 148
    11.3.2 On Graph Partitioning ............................................. 149
    11.3.3 On Data Query Languages .................................... 149
  11.4 Afterword & Lessons Learnt .......................................... 150
Contents

A SPARQL Queries Used in Statistical Data Experiment 153

B Bibliometrics SPARQL Queries Used in Evaluating Gagg 155

Bibliography 163
## List of Figures

1.1 A layered architecture of big data processing systems (thesis contributions are marked by a dashed border) .................................. 8  
2.1 Example RDF graph ................................................. 11  
2.2 Tree representation of an example SPARQL query ................. 14  
2.3 Example MapReduce job to count predicates in RDF data ........ 16  
3.1 \texttt{bAccess} example (absence of graph components indicates that they are empty) ...................................................... 32  
3.2 \texttt{PROJ} example .................................................... 32  
3.3 \texttt{bEXT} example ............................................... 33  
3.4 \texttt{gEXT} example .................................................... 33  
3.5 \texttt{MATCH} example .................................................. 34  
3.6 \texttt{CROSS} example ............................................... 34  
3.7 Illustrative representation of operators that transfer values between an expression (left) and the result of applying an operator to it (right) 38  
3.8 Pictorial representation of expressions used in proving independence of \texttt{FILTER} operator ................................................. 39  
3.9 \texttt{gAccess} example (absence of binding components indicates that they are empty) ...................................................... 40  
3.10 \texttt{E-JOIN} example ................................................. 41  
3.11 Evaluation of the \texttt{GMATCH} expression: $e[((?,p1,?), (?,p2,?)), 2 = 3]$ ............................................................... 43  
4.1 A proof tree that shows that $\leq$ is transitive ...................... 53  
4.2 Analysis of the dependency of \texttt{RDF.co} expressions values. $g$ and $b$ are the graph and binding components of the input while $g'$ and $b'$ are the graph and binding components of the result of applying the expression to the input ................................................. 62  
5.1 \texttt{AGG} example .................................................... 69  
5.2 Operational flow of the graph aggregation ........................... 70  
5.3 \texttt{Gagg} example ................................................... 71  
5.4 Diagrammatic representation of applying \texttt{bfilter} function as part of \texttt{Gagg} evaluation in Figure 5.3 ............................................ 72  
5.5 Aggregation operators analysis ....................................... 74  
6.1 Example social network ............................................... 80
6.2 Example partitioning of the graph in Figure 6.1 into star subgraphs 81
6.3 Example partitioning of the graph in Figure 6.1 into 2-hops subgraphs 82
6.4 Example partitioning of the graph in Figure 6.1 into star subgraphs
and additionally including people who share a common friend . . . . 84
7.1 Example social network . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
7.2 Example partitioning pattern and queries . . . . . . . . . . . . . . . 90
7.3 Example partitioning of the graph in Figure 7.1 using the pattern
in Figure 7.2a . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
7.4 Example decomposition of a query Q given a partitioning pattern P 99
8.1 Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
8.2 Statistical data structure based on the Data Cube vocabulary . . . 104
8.3 LUBM - Distribution of partitions sizes (small horizontal line is the
mean of partitions sizes) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
8.4 LUBM experiment - Query processing times . . . . . . . . . . . . . . 108
8.5 Statistical data experiment - Query processing times . . . . . . . . 108
9.1 Overview of SYRql Implementation . . . . . . . . . . . . . . . . . . . . 118
9.2 A naive execution plan for the expression (((e_1 \times e_2)[(s,p,?)][1 =
"a"]\).b (the final result is indicated by a dashed box) . . . . . . . . . 119
9.3 An optimised execution plan for the expression (((e_1 \times e_2)[(s,p,?)][1 =
"a"]\).b . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
9.4 Query processing times . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
10.1 Bibliographic network inspired from [145]. Dashed nodes represent
papers, and dotted nodes the authors . . . . . . . . . . . . . . . . . . . . . 126
10.2 Operational flow of the graph aggregation . . . . . . . . . . . . . . . . 127
10.3 Summary graphs of the data in Figure 10.1. The number within
parentheses report a count statistics of the aggregated data . . . . . . 129
10.4 The grouped graph that corresponds to the aggregated graph in
Figure 10.3b. This graph aggregates authors in Figure 10.1 by their
organisation and position . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
11.1 (Reproduced from Chapter 1) A layered architecture of big data
processing systems. Thesis contributions are marked by a dashed
border . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
List of Tables

2.1 Results of two example SPARQL triple patterns and their join on the data in Figure 2.1 ........................................... 13
2.2 A comparison of a number of data query languages .................. 23
6.1 Number of triples sent over the network while query answering using different partitioning approaches .......................... 85
7.1 The result of evaluating the path ?x, friend, ?z, ^friend, ?y against the graph of Figure 7.1 using Algorithm 1 (friend is abbreviated as :fr). The first column matches sP_root and determines, therefore, the subgraph to which the triples will be added .......... 94
8.1 LUBM - Loading time and data redundancy .......................... 105
8.2 Statistical data - Loading time and data redundancy ................. 105
8.3 LUBM - Comparison of the size of data between random placement (-rand) and clustering (-clust) ................................. 106
8.4 LUBM - Query decomposition .......................................... 107
8.5 Statistical data - Query decomposition .................................. 107
10.1 Average running times of type summary queries ...................... 138
10.2 Average running times of bibliometrics queries on the SP2B data 138
11.1 Summary of RDF.co operators characteristics ......................... 145
To my wife Gofran
Chapter 1

Introduction

The cost of acquiring and storing data has dropped dramatically in the last few years. Consequently, petabytes and terabytes datasets are becoming commonplace, especially in industries such as telecommunication, health care, retail, pharmaceutical and financial services\(^1\). This collected data is playing a crucial role in societies, governments and enterprises. For instance, data science is increasingly utilised in supporting data-driven decisions and in delivering data products [107, 136]. Furthermore, scientific fields such as bioinformatics, astronomy and oceanography are going through a shift from “querying the world” to “querying the data” in what commonly is referred to as e-science [78]. The main challenge nowadays is analysing the data and extracting useful insights from it.

In order to process the available massive amount of data, a number of distributed frameworks were built to run on top of a cluster of commodity machines. In 2004, Google introduced the MapReduce framework [46] and its open source implementation, Hadoop\(^2\), came out in 2006. Microsoft also introduced Dryad [83], its own distributed computation engine and Apache Spark\(^3\) came out of the Berkeley AMP Lab. Furthermore, there has also been a surge of activities on layering distributed and declarative programming languages on top of these platforms. Examples include PIG Latin from Yahoo [125], DryadLINQ from Microsoft [174], Jaql from IBM [16], HiveQL from Facebook [157], and Meteor/Sopremo [77].

\(^1\)According to a report from IBM in 2013, 90% of the data in the world today has been created in the last two years alone. This report is available at: [https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html](https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html)

\(^2\)[http://hadoop.apache.org](http://hadoop.apache.org)

\(^3\)[http://spark.apache.org](http://spark.apache.org)
As part of this big data movement, richer data models are being adopted to glean stronger signals from the available data and to integrate heterogeneous data sources. In this context, RDF [44], the graph-based data model of the Semantic Web, is increasingly utilised. Consequently, the size of available RDF data is increasing and massive datasets are becoming commonplace. The 1.2 billion triples of Freebase can be freely downloaded\(^4\) and the LOD Cloud grew to 31 billion RDF triples as of September 2011.\(^5\) Furthermore, distributed execution platforms are being utilised to process RDF data; for both query answering [34, 81, 143] and reasoning [106, 165, 166]. However, there has not been much activity in introducing high-level languages to support RDF analytics and processing.

When analysing large RDF datasets, users are left mainly with two options: using SPARQL [72], the main query language for RDF, or using an existing non-RDF-specific big data language. This thesis argues that each of these two approaches has its own limitations. We, therefore, pursue defining a dataflow language specifically designed to process RDF data on top of distributed platforms. Using a dataflow language, a program is conceptually a directed graph where nodes are operations and edges represent data flowing between them. Operations are executed as soon as their operands become available. Thus, dataflow languages are inherently parallel and can work well in decentralised systems [45, 86].

### 1.1 Thesis Context

In developing a dataflow language in this thesis, we discuss three components: (i) the data query language, (ii) the underlying data model, and (iii) the physical arrangement of the underlying distributed data. Figure 1.1 shows the layout of the different components together with some popular distributed computation engines. It is worth pointing out that these three components are independent from the distributed computation engine and can, therefore, be plugged into different platforms.

\(^4\)https://developers.google.com/freebase
\(^5\)http://lod-cloud.net/state/
1.1.1 On Data Query Languages

Declarative query languages have been a corner stone of data management since the early days of relational databases. The initial proposal of relational algebra and relational calculus by Codd [37] was shortly followed by other languages such as SEQUEL [25] (predecessor of SQL) and QUEL [153]. Declarative languages simplified programming and reduced the cost of creation, maintenance, and modification of software. They also helped bringing the non-professional user into effective communication with a database [40]. Database languages design continued to be an active area of research and innovation. In 2014, the Beckman Report on Database Research identified declarative programming as one of the main research opportunities in the data management field [3].

It is increasingly recognised that SQL, the main database query language, has a number of limitations that restrict its utility in analytics and complex data processing scenarios [12, 47, 125, 154, 174]. One limitation of SQL is its purely declarative nature. According to [125], “programmers often find it unnatural to analyse data by writing pure declarative queries in SQL instead of writing imperative scripts”. Limitations of SQL resulted in developing a number of languages that is usually referred to as NoSQL languages [24]. Some of these languages, such as HiveQL, Pig Latin, and Jaql are particularly designed to suit distributed platforms.

For RDF data, SPARQL [72] is the main query language and is the one recommended by the W3C. SPARQL is a graph pattern matching language that provides rich capabilities for slicing and dicing RDF data. The latest version, SPARQL 1.1, supports also aggregation and nested queries. Nevertheless, it can be argued that SPARQL has limitations similar to those of SQL. SPARQL’s purely declarative nature obligates a user to express their needs in a single query. This can be unfamiliar for some programmers and challenging for complex needs [68, 79, 111, 167]. Furthermore, SPARQL evaluation is known to be costly [129, 146]. Consequently, a “one size does not fit all” belief is increasingly present about querying semantic web data. Linked Data Fragments [167] and link traversal [76, 164] are two prominent examples of new querying approaches. This thesis explores the direction of a simple dataflow language that is suitable for distributed systems.

The other alternative for querying large RDF data is using an existing big data language, such as Pig Latin or HiveQL. We argue that this approach also has
its own limitations. These languages were designed for tabular data mainly, and, consequently, using them with RDF data commonly results in verbose, unreadable, and sometimes inefficient scripts. For instance, listings 1.1 and 1.2 show a basic SPARQL graph pattern and an equivalent Pig Latin script, respectively. Listing 1.2 has double the number of lines compared to listing 1.1 and is, arguably, harder to read and understand.

Listing 1.1: SPARQL basic pattern

```sparql
?prod a :ProductType .
?p :reviewer ?rev
```

Listing 1.2: Corresponding Pig Latin script

```pig
rdf = LOAD 'data' USING PigStorage(' ') AS (S,P,O);
SPLIT rdf INTO reviewers IF P = ':reviewer',
reviews IF P = ':reviewFor',
prods IF P = 'a' and O = 'ProductType';
tmp1 = JOIN prods BY S, reviews BY O;
tmp2 = JOIN tmp BY reviewers::S, reviewers BY S;
```

1.1.2 On Data Models

A data model consists of a notation to describe data and a set of operations used to manipulate that data [163]. The definition of the relational model was a cornerstone in the revolution that redefined the data management field in the late seventies. In an early paper [39], E. F. Codd stated a number of roles a data model provides: (i) as a basis for developing a general design methodology for databases, (ii) as a basis for the development of families of very high level languages for query and data manipulation, (iii) as a focus for DBMS architecture, and (iv) as a vehicle for research into the behavioural properties of alternative organisations of data.

In fact, data models continue to play a crucial role in recent innovations in the database field. The NoSQL movement is an innovation over the relational model as much as it is an innovation over SQL, the language [12, 24]. This resulted in a proliferation of data models which include simple key-value, document-based, JSON-based, and array-based models.

The data model underlying SPARQL is the SPARQL algebra [72]. The SPARQL algebra is defined on sets of bindings (aka mappings). The formal definition of the SPARQL algebra and its expressivity and complexity have been extensively studied [72, 129, 134]. However, the SPARQL algebra cannot be used as a basis for

---

6 We provide more details on the SPARQL Algebra in Section 2.1.1.
a dataflow language, as the SPARQL algebra is not fully composable. SPARQL algebra transitions, via triple pattern matching, from graphs (i.e., the initial inputs) to sets of bindings (which are basically tables resulting from pattern matching). Subsequently, further operators such as Join, Filter, and Union are applied on sets of bindings. In other words, the flow is partly “hard-coded” in the SPARQL algebra and a user cannot, for instance, apply a pattern matching on the results of another pattern matching or “join” two graphs. In a dataflow language, the dataflow is guided by the user and cannot be limited to the way the SPARQL algebra imposes (Section 2.1 provides further discussion of this limitation).

1.1.3 On Managing Graph Data

The flexibility of the graph data model makes it an attractive integration vehicle for data from various domains and sources. As a result, the size and complexity of existing graphs are growing. Examples can be seen in graph-based knowledge bases such as WikiData\(^7\) and DBpedia\(^8\), in graphs used by Internet companies such as the Google Knowledge Graph\(^9\) and the Facebook Social Graph\(^10\), and in graphs used in bio-informatics\(^11\).

Graph data poses unique computational challenges due to their unstructured nature and high data access to computation ratio [109]. This becomes particularly challenging in distributed environments. Consequently, there exists a rich body of research on graph-specific distributed systems [63, 108, 114, 172], graph partitioning algorithms [91], graph data serialisation and compression [55, 95], and graph-specific parallelisation techniques [133].

In this thesis, we focus only on the graph partitioning problem. We contend that the partitioning algorithms used in distributed graph systems, such as ParMETIS [90] and balanced vertex-cut [63] algorithms, do not match the heterogeneity of recent graphs. These algorithms are designed for unlabelled graphs. However, RDF graphs are edge-labelled directed multi-graphs. In RDF graphs, different edges have different importance in different scenarios. Furthermore, related entities are

\(^7\)http://www.wikidata.org
\(^8\)http://dbpedia.org
\(^9\)http://googleblog.blogspot.fr/2012/05/introducing-knowledge-graph-things-not.html
\(^10\)https://developers.facebook.com/docs/graph-api
not necessarily linked by direct edges. In this thesis, we work on defining an RDF graph partitioning approach that considers graph structure and edge labels and that can leverage knowledge that might be available about the data or the task at hand.

1.2 Thesis Approach and Hypotheses

1.2.1 On Managing Graph Data

Our approach is based on using pattern matching, the main component for querying graphs, to guide graph partitioning. This approach has a number of benefits; (i) it accommodates graph structure and edge labels, (ii) it unifies partitioning definition and queries based on pattern matching, and (iii) it allows utilising available knowledge about the data and the query load.

Hypothesis 1. When prior knowledge is available about query load, partitioning an RDF graph using pattern matching enhances performance of query answering.

While pattern matching can be useful to query any graph, it is arguably easier and more useful for domain-specific graphs. Such domain-specific graphs have recurring patterns that make it easier to meaningfully query them. Our work on partitioning focuses on domain-specific graphs as reflected by the graphs used later in the experimental evaluations (e.g., we use LUBM graphs from the education domain and graphs representing statistics).

1.2.2 On Data Models

The transition from graphs to sets of bindings in the SPARQL algebra introduces some restrictions on composing and cascading triple pattern matching and other operators. Our approach to define a fully composable data model is to associate two components with each expression value: a binding and a graph. Moreover, such data model should have a small set of operators that are amenable to parallelisation.
**Hypothesis 2.** Pairing bindings and graphs in expression values allows defining a fully composable data model for querying RDF data. The algebraic properties of this model provide optimisation opportunities.

### 1.2.3 On Data Query Languages

A declarative language can be defined on top of the defined data model. Such a language enables writing dataflow scripts that get transparently translated and executed on a distributed platform. We focus on the MapReduce platform in the thesis.

**Hypothesis 3.** A dataflow language for big RDF data allows declarative manipulation of RDF data and provides a similar performance to existing big data languages when implemented on top of the MapReduce platform.

### 1.3 Contributions and Thesis Outline

The main contributions of this thesis are organised in three parts. The first part describes the data model we developed as a basis for a dataflow language. The second part describes our work on partitioning RDF graphs based on pattern matching. The third part describes our work on SYRql, a dataflow language. The third part also describes a proposal to extend SPARQL with a graph aggregation operator that is part of RDF.co, the data model described in this thesis. The next subsections highlight the main contributions which are also shown in Figure 1.1.

### 1.3.1 On Data Models

We introduce RDF.co, a data model that defines a pair of a binding and a graph in the value of each expression. RDF.co, similar to other big data models, defines a small set of basic operators that are amenable to parallelisation. In summary, our contributions on data models are:

- A formal definition of the syntax and semantics of the model is presented in Chapter 3 [Hypothesis 2].
• We characterise the expressivity of RDF.co by comparing it to SPARQL. Some interesting parallels to and differences from SPARQL are also discussed in Chapter 3 [Hypothesis 2].

• Chapter 4 discusses a number of unique algebraic properties of the data model that can be used in query optimisation [Hypothesis 2].

• Chapter 5 describes aggregation operators in RDF.co. A graph aggregation operator, an operator that has no equivalence defined in SPARQL, is also described [Hypothesis 2].

In particular, we believe that the algebraic properties in Chapter 4 represent a unique study on relations between triple patterns. Our hope is that these results prove useful in related fields such as SPARQL query optimisation and view-based query answering.

### 1.3.2 On Graph Partitioning

Our contributions on graph partitioning are:
Chapter 6 presents a generalised formulation of the problem of graph partitioning. This formulation can define pattern-based partitioning as well as other existing partitioning methods [Hypothesis 1].

Chapter 7 presents a formal model and algorithms to realise pattern-based graph partitioning and to utilise the partitioning during query answering [Hypothesis 1].

An implementation and an experimental study is presented in Chapter 8 [Hypothesis 1].

In particular, we believe that the main result is mapping query answering over partitioned graphs to the well-studied problem of view-based query answering. The relevant theorem (Theorem 7.1), which defines graph-centric criteria sufficient for query containment, can be useful in other scenarios such as the use of cached query results and routing (sub)queries in a federated query processing scenario.

1.3.3 On Data Query Languages

Our contributions on data query languages are:

- Chapter 9 describes SYRql, a dataflow language for large scale processing of RDF data. An implementation of SYRql on top of the MapReduce platform and a corresponding evaluation are also presented [Hypothesis 3].

- Chapter 10 describes extending the SPARQL algebra by a graph aggregation operator similar to that defined as part of RDF.co. Corresponding implementation and evaluation results are also presented.

1.3.4 Summary of Publications

We motivated the need for analytics language for distributed processing of RDF data in a paper presented in Consuming Open Linked Data Workshop (COLD 2013) [111]. SYRql was presented in ISWC 2014 [112] and nominated for best student paper award. Our work on graph aggregation for SPARQL was presented in ESWC 2015 [110] and won the best student paper award.
1.4 Research Methodology

The study of the data model, $\text{RDF.co}$, is achieved by formally defining the model using mathematical logic. This is similar to the way relational algebra is defined [5]. Furthermore, the SPARQL algebra is also described using the same approach [72, 129].

Formal definition of data models allows precise and concise description of the different constructs. Moreover, it allows studying the interaction between these constructs to identify any inconsistency, redundancy, or contradiction. It is common to study the expressive power of a data model by mapping it to another well-studied model. For example, the SPARQL algebra was studied by comparing it to the well-studied datalog that underlies relational algebra [8, 134]. Similarly, we map $\text{RDF.co}$ to a subset of the SPARQL algebra.

We provide a formal definition of pattern-based graph partitioning. For SYRql, the dataflow language, we use the Backus-Naur Form notation to define its syntax while the semantics is based on the definition of the underlying data model, $\text{RDF.co}$. Finally, we use benchmarking to evaluate our implementations of both pattern-based partitioning and SYRql.
Chapter 2

Preliminaries

In this chapter we provide a short background on RDF, SPARQL and existing big data platforms. We further motivate our work to go beyond the existing SPARQL algebra and discuss related efforts on big data and RDF querying languages.

2.1 RDF & SPARQL

The Resource Description Framework (RDF) [44, 94] is a data model for representing information about World Wide Web resources. RDF data is a set of triples where each triple is a simple statement about a resource. A triple takes the form of subject, predicate, object. RDF was published as a W3C Recommendation in 2004 and a new revised version, RDF 1.1, was then published in 2014. While RDF data can be serialised in different syntaxes, we will use Turtle\(^1\) throughout the thesis. Furthermore, RDF data can be viewed as a graph where each triple represents a labelled edge between two vertexes (from subject to object). Figure 2.1 shows a graph representation of an example RDF data serialised in Turtle in Listing 2.1.

---

1. [http://www.w3.org/TR/2014/REC-turtle-20140225/](http://www.w3.org/TR/2014/REC-turtle-20140225/)

---

\[ \text{Figure 2.1: Example RDF graph} \]
While RDF was proposed initially to encode knowledge in the Semantic Web, its application has expanded to the wider discourse of computer science. For instance, RDF is used to represent bio-informatics\(^2\) and government\(^3\) data. Furthermore, RDF is used in enterprises to integrate heterogeneous data\(^4\).

For querying RDF data, SPARQL is the language recommended by the W3C. SPARQL is a graph pattern matching language that provides rich capabilities for slicing and dicing RDF data. The latest version, SPARQL 1.1, supports also aggregation and nested queries.

### 2.1.1 The SPARQL Algebra

The formal presentation of SPARQL was reported in [129] and is included as part of the SPARQL W3C specifications. In this section, we present the formal definition of only one operator of SPARQL, namely the AND operator. The goal is just to provide enough details to discuss composable of SPARQL operators (i.e., the ability to cascade operators in order to obtain more complex expressions). Our presentation here is based on the one in [129].

Assume the two pairwise disjoint infinite sets \(I\) (IRIs) and \(L\) (literals). \(T = I \cup L\) is the set of RDF terms. A triple \((s, p, o) \in I \times I \times (I \cup L)\) is called an RDF triple\(^5\). In this tuple, \(s\) is the subject, \(p\) the predicate and \(o\) the object. Assume additionally the existence of an infinite set \(V\) of variables disjoint from the above sets.

An AND SPARQL graph pattern expression is defined recursively as follows\(^6\):

\(^2\)E.g., \url{http://data.gov.uk}
\(^3\)E.g., \url{http://bio2rdf.org}
\(^4\)Example of companies that use RDF include BBC, Telefonica, Novartis and Daimler.
\(^5\)We only consider ground RDF graphs and therefore we do not consider blank nodes.
\(^6\)In addition to AND, SPARQL defines UNION, OPT, and FILTER expressions.
1. A tuple from \((\mathcal{I} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{L} \cup \mathcal{V})\) is a graph pattern (a triple pattern).

2. If \(P_1\) and \(P_2\) are graph patterns, then expression \((P_1 \text{ AND } P_2)\) is also a graph pattern.

A mapping \(\mu\) from \(\mathcal{V}\) to \(\mathcal{T}\) is a partial function \(\mu : \mathcal{V} \rightarrow \mathcal{T}\). The domain of \(\mu\), \(\text{dom}(\mu)\), is the subset of \(\mathcal{V}\) where \(\mu\) is defined. Two mappings \(\mu_1\) and \(\mu_2\) are compatible when for all \(x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)\), it is the case that \(\mu_1(x) = \mu_2(x)\), i.e., when \(\mu_1 \cup \mu_2\) is also a mapping.

Let \(\Omega_1\) and \(\Omega_2\) be sets of mappings. We define the join of \(\Omega_1\) and \(\Omega_2\) as:

\[
\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}
\]

Given an RDF graph \(G\), we define the semantics of \(\text{AND}\) graph pattern expressions as a function \(\llbracket \cdot \rrbracket_G\) which takes a pattern expression and returns a set of mappings.

1. For a triple pattern \(t\), \(\llbracket t \rrbracket_G = \{\mu | \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in G\}\), where \(\text{var}(t)\) is the set of variables occurring in \(t\) and \(\mu(t)\) is the triple obtained by replacing the variables in \(t\) according to \(\mu\).

2. For two graph patterns \(P_1\) and \(P_2\), \(\llbracket (P_1 \text{ AND } P_2) \rrbracket_G = \llbracket P_1 \rrbracket_G \bowtie \llbracket P_2 \rrbracket_G\).

Table 2.1 shows an example of two SPARQL triple patterns along with the expression that joins them together using \(\text{AND}\). Sets of mappings are represented using tables with variable names in the header.

<table>
<thead>
<tr>
<th align="left">?s :livesIn ?o</th>
<th align="center">?o :partOf ?c</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left">:John</td>
<td align="center">:Galway</td>
</tr>
<tr>
<td align="left">:Ronan</td>
<td align="center">:Dublin</td>
</tr>
<tr>
<td align="left">:Galway</td>
<td align="center">:Dublin</td>
</tr>
<tr>
<td align="left">:Dublin</td>
<td align="center">:Ireland</td>
</tr>
</tbody>
</table>

\((?s :\text{livesIn } ?o) \text{ AND } (?o :\text{partOf } ?c)\)

<table>
<thead>
<tr>
<th align="left">?s</th>
<th align="center">?o</th>
<th align="center">?c</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left">:John</td>
<td align="center">:Galway</td>
<td align="center">:Ireland</td>
</tr>
<tr>
<td align="left">:Ronan</td>
<td align="center">:Dublin</td>
<td align="center">:Ireland</td>
</tr>
</tbody>
</table>

Table 2.1: Results of two example SPARQL triple patterns and their join on the data in Figure 2.1
2.1.2 Composability of the SPARQL Algebra

The SPARQL algebra is defined on sets of mappings. Each of its main operators, \texttt{AND}, \texttt{UNION}, and \texttt{OPT} takes a set of mappings as input and produces a set of mappings as a result. Therefore, these operators can be arbitrarily cascaded and SPARQL is fully composable. However, this does not apply to triple pattern matching. Triple pattern matching is evaluated against a graph to obtain a set of mappings. Only after applying triple pattern matching, the rest of operators can be applied.

It is common to represent queries as trees where each node is an operator and its children are the inputs of the operator. This representation is commonly used for relational algebra expressions and was also used for SPARQL queries (see \cite{75} for an example). Figure 2.2 provides a tree representation of the sample query shown before (in Table 2.1). For a SPARQL query, leaves are always triple pattern matching operators. Furthermore, triple pattern matching operators cannot be internal nodes in the tree. In comparison, leaves in relational algebra expressions can be any of the relational algebra operators. Therefore, we state that the flow is partially hard-code in a SPARQL query. Evaluation always starts with applying triple patterns that allows the transition from graphs to sets of mappings. Afterwards, further operators can be applied to the sets of mappings but no triple pattern matching can be applied on these sets.

2.1.3 Complexity of SPARQL Evaluation

It has been shown that the evaluation of SPARQL queries is PSPACE-complete \cite{129}. In \cite{146}, it was shown that the SPARQL operator \texttt{OPTIONAL} alone is responsible for the PSPACE-completeness of the evaluation problem. Furthermore, \cite{129} proved that the evaluation is NP-complete for graph pattern expressions constructed by using only \texttt{AND}, \texttt{FILTER} and \texttt{UNION} operators.
2.2 Big Data Systems

It was reported that data is growing faster than Moore’s law\(^7\). Consequently, parallel processing of data is increasingly utilised to keep up with the fast growth of data. Parallel processing can be achieved by scaling up or scaling out. Scaling up is based on adding more cores to a machine, whereas scaling out refers to the use of multiple commodity machines connected together. For scalability and economic reasons, scaling out approach is increasingly adopted.

In 2004, Google introduced the MapReduce framework [46] and its open source implementation, Hadoop\(^8\), came out in 2006. MapReduce became very popular as a general purpose distributed platform. It was used to process both unstructured and structured data, including massive graph data [48, 81, 88]. Afterwards, distributed platforms designed specifically for graph data were proposed. We discuss both the MapReduce and the massive graph distributed systems next.

2.2.1 The MapReduce Model

The MapReduce model decomposes a program into a set of jobs. Each job consists of a number of tasks that run in parallel on different machines close to where the data is stored. Each task composes two steps: a map and a reduce. The output of each map is a list of pairs of keys and values. Before invoking the reduce tasks, the framework groups the output of the map tasks by their keys. It then shuffles the data so that all pairs that share key value are processed by the same reduce task. The framework abstracts the complicated low-level details such as scheduling, fault tolerance, and network communication from the user.

A MapReduce program can, therefore, be defined by two functions:

- **Map:** \((k1,v1) \rightarrow \text{list}(k2,v2)\)
  (i.e., a map function takes a pair \((k1,v1)\) as input and outputs a list of other pairs of the form \((k2,v2)\))

- **Reduce:** \((k2, \text{list}(v2)) \rightarrow \text{list}(v3)\)
  (i.e., a reduce function takes the output pairs \((k2,v2)\) of the map function grouped by their key and outputs a list of new values \(v3)\)

\(^8\)http://hadoop.apache.org
Listing 2.2 shows an example pseudo code for a MapReduce program that counts the occurrences of each predicate in some RDF data. Figure 2.3 provides a diagrammatic representation of the execution of the MapReduce program on the example RDF data shown before. In the example, the map function is of the form \( (\text{key}, \text{triple}) \rightarrow \text{List}((\text{predicate}, \text{count})) \) where the key of the input is not used. The reduce function in Listing 2.2 is of the form \( (\text{predicate}, \text{List}((\text{count}))) \rightarrow \text{List}(\text{counts}) \).

```java
function map(String triple):
    parse triple into (s,p,o)
    emit (p, 1)

function reduce(String predicate, Iterator partialCounts):
    count = 0
    for each pc in partialCounts:
        count += pc
    emit (predicate, count)
```

**Listing 2.2:** A MapReduce pseudo code that counts the occurrences of each predicate in some RDF data

![Diagram of MapReduce execution](image)

**Figure 2.3:** Example MapReduce job to count predicates in RDF data

### 2.2.2 Vertex-centric Model

In 2010, Pregel was described in a paper by Google researchers [114]. Pregel adopts a vertex-centric approach in which programs are expressed as a sequence of iterations. In each iteration, a vertex collects messages sent in the previous
iteration, modifies its own state, and then sends messages along its outgoing edges. Apache Giraph\(^9\) is an open source implementation of Pregel. Both systems are inspired by the Bulk Synchronous Parallel model of distributed computation that divides a program into a number of sequential supersteps. Communication takes place only between supersteps in order to ensure scalability and fault tolerance.

A number of systems similar to Pregel were presented such as GraphLab [108], PowerGraph [63] and GraphX [172]. Signal/Collect [155] approach is worth being highlighted as it was designed specifically for RDF graphs. Similar to Pregel, Signal/Collect is a vertex-centric approach that is based on Bulk Synchronous Parallel model. However, Signal/Collect supports asynchronous computations and different types of vertices in the graph.

Furthermore, there has been a number of proposals to extend the “think like a vertex” model of Pregel to a more general “think like a graph” model [137, 147, 158, 173]. All these platforms are mainly designed for iterative algorithms such as computing PageRank and finding strongly connected components in massive graphs. To the best of our knowledge, they have been only seldom used for pattern matching (e.g., [54]). Nevertheless, approaches to distribute the graph data in these systems are relevant to our work on pattern-based graph distributing (Chapter 6).

### 2.3 Big Data Systems & RDF

Big data systems, particularly MapReduce, have been utilised to support distributed processing of SPARQL queries over large amounts of RDF data. Some systems translate SPARQL queries directly into a set of MapReduce jobs [122, 139]. These systems optimise performance through reducing the number of required MapReduce jobs (i.e., coalescing operators) [122, 138, 139], using heuristics to order required joins [122], and building local indices [62]. Other systems, such as RAPID [138] and PigSPARQL [143], translate SPARQL queries into another big data language, such as Pig Latin. The resulting script is then translated into a series of MapReduce jobs. Furthermore, [81] deployed a state-of-the-art RDF store on each node in the cluster and used MapReduce only when results from different machines must be joined. [59] presented a generic algorithm to execute a SPARQL

\(^9\)http://giraph.apache.org/
basic graph pattern on MapReduce systems. A survey of SPARQL querying on top of MapReduce systems was recently published [60]. It is worth mentioning that most existing systems support only SPARQL basic graph patterns.

Besides being used for query processing, big data systems have also been used to compress RDF data [61] and to reason over it [106, 165, 166].

### 2.4 Dataflow Languages Definition

The original motivation for research into dataflow was the exploitation of massive parallelism. While earlier work on dataflow focused on hardware design and proposed dataflow machines as an alternative to the prevalent von Neumann architecture, more recent works focused on the language aspects of dataflow programming [86]. Conceptually, a dataflow program is a graph of operators with data flowing between them. An operator gets executed once its input data is available (i.e., the flow is controlled by the data). A dataflow language typically supports a list of features [86]:

1. Freedom from side effects
2. Data dependencies equivalent to scheduling
3. Single assignment of variables

Recent research on parallel and distributed platforms revived the interest in dataflow languages for processing big datasets [1, 26, 121, 125].

### 2.5 Existing Data Languages

In this section, we describe a number of languages that are defined mainly to query data. We focus on languages designed for big data systems, for graph data, and for RDF data. For each language, we report its underlying data model, the set of operators it supports, and its programming paradigm. The main data models for the studied languages are the relational model, RDF, property graph, and, for the data languages embedded in general-purpose programming languages,
the object-oriented data model. Programming paradigm refers to the style of expressing queries in the language. When it comes to programming paradigm, the three main categories we observe are: (i) imperative, where queries describe how results should be achieved, (ii) declarative, where queries describe what results are required rather than how to achieve them, and (iii) dataflow, where a query is a graph of functional operators with data flowing between them. The distinction between these different categories are not a clear cut as these distinctions rely on the context within which a language is studied and because languages tend to mix concepts from different paradigms.

2.5.1 Big Data Languages

**Pig Latin.** Pig Latin [125] is a dataflow language defined on top of MapReduce. A program in Pig Latin is modelled as a directed graph of the data flowing between operations. Operations in Pig Latin are similar to those defined in relational algebra. However, different from SQL, Pig Latin is not a purely declarative language. Pig Latin is commonly used for extract, transform and load (ETL) workload.

**HiveQL**. HiveQL [157] is a SQL-like query language of Apache Hive\(^\text{10}\). HiveQL is very similar to SQL, however its implementation is based on MapReduce.

**Jaql**. Jaql [16] is a dataflow language for analysing large semistructured datasets in parallel using MapReduce. Jaql is similar to Pig Latin as it defines a number of operators equivalent to those defined in relational algebra. Also being a dataflow language, a Jaql program is modelled as a directed graph of the data flowing between operations. Nevertheless, the data model underlying Jaql is JSON.

**FlumeJava**. FlumeJava [26] is a Java API that allows pipelining MapReduce jobs. FlumeJava uses deferred evaluation which allows it to perform a number of optimisations before the actual execution of the MapReduce jobs.

**Apache Spark**. Apache Spark [1] originates from a project in the Berkeley AMP Lab. The main abstraction of Spark is the Resilient Distributed Dataset (RDD), a fault-tolerant distributed data structure [175]. A Spark program is a series of distributed transformations and actions applied to the RDDs. Spark is distributed as a Scala library but can also be used from Python and R.

\(^{10}\)https://hive.apache.org/
2.5.2 RDF Data Languages

As described before, SPARQL is the main language to query RDF data. A survey of RDF querying languages prior to SPARQL is presented in [57]. Moreover, there has been a large number of proposed extensions to SPARQL. For instance, the ρ operator added support for exploratory search for semantic associations [10], nSPARQL added nested regular expressions [130], and [97] proposed adding a CLUSTER BY statement to SPARQL to allow clustering results. Nevertheless, all these extensions did not change the pure declarative nature of SPARQL. In addition to SPARQL, we discuss a number of other languages that were defined to query RDF data.

**LDQL.** LDQL [76] is a declarative language to query Linked Data on the Web. LDQL queries define a SPARQL query to describe the results, but also define navigation paths that select the data sources to be used. The data model underlying LDQL is RDF data model (for the SPARQL part) and a model of the Web of Linked Data (for the web paths part).

**XSPARQL.** XSPARQL [17] is a scripting language that combines SPARQL and XQuery in one language to integrate and query data. It was originally proposed as a W3C member submission in 2009. XSPARQL supports RDF-to-RDF transformation, lowering (RDF-to-XML), and lifting (XML-to-RDF). The programming paradigm of XSPARQL is a combination of declarative pattern matching and FLOWR expressions (For-Let-Where-Order-Return).

**LITEQ.** LITEQ [150] embeds RDF processing in a programming language. It implements NPQL, a novel path query language, that provides a simple graph traversal based query language that gets translated into SPARQL queries. Returned results of these queries are automatically typed in the programming language. LITEQ is embedded in F# and is, therefore, based on an object oriented data model.

Similar to LITEQ, there is also a number of non-declarative languages that can integrate with common programming languages to provide support for RDF data manipulation. Examples include ActiveRDF\(^{11}\) for Ruby and SuRF\(^{12}\) for Python.

\(^{11}\)http://activerdf.org
\(^{12}\)https://github.com/cosminbasca/surfrdf
2.5.3 Graph Languages

While there exists a large number of graph query languages [170], we discuss only two examples here.

**Cypher.** Cypher\(^{13}\) is a declarative pattern matching language. It is the main language to query Neo4j\(^{14}\), a popular graph database. In addition to pattern matching, Cypher supports manipulating the graph by creating new nodes, edges, or updating their properties. The data model underlying Cypher is property graph\(^{15}\).

**Help.** Help \(^{141}\) provides a set of high-level primitives that are commonly used in vertex-centric programs. These primitives can be grouped into three areas; vertex-centric updates, graph topology modification, and global aggregation. Help is implemented on top of Apache Spark GraphX. In this regards, Help can also be considered a big data language.

2.5.4 NoSQL Systems

NoSQL systems have been used to query RDF data \(^{42}\). From the language perspective, NoSQL systems support for graphs vary. We can distinguish four approaches:

- Support SPARQL through some existing RDF library running on top the NoSQL system. In this approach, the NoSQL system acts as only a storage layer. Examples include CumulusRDF\(^{16}\) which runs RDF Sesame\(^{17}\) on top of Apache Cassandra\(^{18}\). Another example is the use of Apache Jena\(^{19}\) on top of Apache HBase\(^{20}\) in \(^{42}\).

- Use a proprietary extension to SQL. Commonly, such extensions only allow getting tabular results from graph data. SAP Hana support for graph is an example of such approach.\(^{21}\)

---

\(^{13}\)http://neo4j.com/docs/stable/cypher-query-lang.html

\(^{14}\)http://neo4j.com/

\(^{15}\)See \(^{74}\) for a definition of property graphs and a comparison between them and RDF graphs.

\(^{16}\)https://code.google.com/archive/p/cumulusrdf/

\(^{17}\)http://rdf4j.org/

\(^{18}\)http://cassandra.apache.org/

\(^{19}\)https://jena.apache.org/

\(^{20}\)https://hbase.apache.org/

• Support Gremlin\textsuperscript{22}, an imperative language for graph traversal. Examples of NoSQL systems that support Gremlin include OrientDB\textsuperscript{23} and Titan\textsuperscript{24}.

• Provide general purpose API and leave it to users to implement queries. Example of such support is the use of the MapReduce API on top of Couchbase\textsuperscript{25} in [42].

### 2.5.5 Summary and Requirements

Table 2.2 shows a comparison of the languages described before. For each language, Table 2.2 shows the underlying data model, set of operators, and the programming paradigm. Our work aims at defining a dataflow language that is tailored for big RDF data. We summarise the required requirements for this language:

• Programming paradigm: the language should support dataflow paradigm as defined in Section 2.4. Particularly, the language should support defining a dataflow query as a graph of operator and manage the dependency between them in terms of data flowing.

• Data model: the language should support a graph data model. In particular, RDF data model should be supported.

• Supported operators: the language should support a subset of SPARQL operators. In particular, operators required to express select-project-join queries (aka conjunctive queries) should be supported. Furthermore, aggregation operators should be supported. Aggregation is important for summarising big datasets.

Furthermore, we gear the language implementation towards batch oriented and analytical queries. Such queries are non-interactive and typically involve a significant portion of the underlying data in order to achieve the intended results.

\textsuperscript{22}\url{https://github.com/tinkerpop/gremlin/wiki}
\textsuperscript{23}\url{http://orientdb.com/orientdb/}
\textsuperscript{24}\url{http://titan.thinkaurelius.com/}
\textsuperscript{25}\url{https://www.couchbase.com/}
Table 2.2: A comparison of a number of data query languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Data model</th>
<th>Operators</th>
<th>Programming paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>HiveQL</td>
<td>Relational</td>
<td>Fully relational</td>
<td>declarative</td>
</tr>
<tr>
<td>Pig Latin</td>
<td>Relational with nested list</td>
<td>Fully relational</td>
<td>dataflow</td>
</tr>
<tr>
<td>Jaql</td>
<td>JSON (nested tree)</td>
<td>Fully relational</td>
<td>dataflow</td>
</tr>
<tr>
<td>FlumeJava</td>
<td>Object Oriented</td>
<td>Turing complete</td>
<td>imperative</td>
</tr>
<tr>
<td>Apache Spark</td>
<td>Object Oriented</td>
<td>Turing complete</td>
<td>imperative</td>
</tr>
<tr>
<td>SPARQL</td>
<td>RDF model (set of triples)</td>
<td>SPARQL operators</td>
<td>declarative</td>
</tr>
<tr>
<td>XSPARQL</td>
<td>RDF model or XML model</td>
<td>SPARQL operator +XQuery operators</td>
<td>declarative + FLWOR expressions</td>
</tr>
<tr>
<td>LDQL</td>
<td>RDF model (set of triples) + Web of Linked Data</td>
<td>SPARQL operators + paths navigation</td>
<td>declarative</td>
</tr>
<tr>
<td>LITEQ</td>
<td>Object Oriented</td>
<td>Path traversal</td>
<td>imperative</td>
</tr>
<tr>
<td>Help</td>
<td>Graph</td>
<td>graph topology and vertices updates</td>
<td>imperative</td>
</tr>
<tr>
<td>Cypher</td>
<td>property graphs</td>
<td>pattern matching and manipulation</td>
<td>declarative</td>
</tr>
<tr>
<td>Jena + HBase</td>
<td>RDF</td>
<td>SPARQL</td>
<td>declarative</td>
</tr>
<tr>
<td>SAP Hana proprietary SQL</td>
<td>property graphs</td>
<td>pattern matching and aggregation</td>
<td>declarative</td>
</tr>
<tr>
<td>Titan and OrientDB</td>
<td>property graphs</td>
<td>Gremlin</td>
<td>imperative</td>
</tr>
<tr>
<td>Couchbase</td>
<td>JSON</td>
<td>Turing complete</td>
<td>imperative</td>
</tr>
<tr>
<td>MapReduce API</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.6 Summary

This chapter provided background and work related to this thesis. The core contributions of this thesis is described in the next three parts of the thesis. The
following parts describe our work on introducing a new data model for RDF processing, a model and implementation of graph partitioning, and, finally, a dataflow language for large RDF data.
Part I

RDF.co, the Data Model
Chapter 3

RDF.co, Composable Operators to Manipulate RDF Data

In this chapter, we introduce RDF.co, a fully composable data model that can be used to manipulate and query RDF data. A data model consists of a notation to describe data and a set of operations used to manipulate that data [163]. We provide a formal definition of a small set of basic operators that constitute RDF.co. Based on this formal definition, we prove that the proposed set of operators is minimal by showing that removing any of these operators reduces the expressivity of the model. We then present a number of derived operators that can be defined using other RDF.co operators. Derived operators provide convenience and brevity to the data model but does not alter neither the expressivity nor the computational complexity of the data model. Finally, we show that the data model can express an important subset of operators that are contained in SPARQL, the language recommended by the W3C to query RDF data. In comparison to the SPARQL algebra, RDF.co is fully composable and allows arbitrary cascading of its operators, a property that is essential for dataflow languages.

1RDF.co Stands for RDF Composable Operators
3.1 RDF.co Expressions

3.1.1 Preliminaries

We use $\mathbb{N}$ to denote the set of all natural numbers that are strictly bigger than 0. We assume the existence of two disjoint countably infinite sets: $\mathcal{U}$ (URIs [15]), $\mathcal{L}$ (literals). $\mathcal{T} = \mathcal{U} \cup \mathcal{L}$ is the set of RDF terms (i.e., $\mathcal{T}$ is the set of all URIs and literals). We also assume the existence of a set $\mathcal{H}$ (function symbols) and a set $\mathcal{C}$ (graph symbols) and assume that $\mathcal{C}$ contains a symbol $g^\phi$ (i.e., $g^\phi \in \mathcal{C}$). We also assume that the sets $\mathcal{U}$, $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{H}$ are disjoint from $\mathbb{N}$. Furthermore, we assume the existence of the symbol ‘?’ such that $\not\in \mathcal{T}$ and define a triple pattern as any triple in $(\mathcal{T} \cup \{\}\} \times (\mathcal{T} \cup \{\}\} \times (\mathcal{T} \cup \{\})$. The set of symbols we use in defining RDF.co expressions, i.e., the alphabet, also contains $\{(), ., b, \divides, \oplus, [., ] , \times, =, \neq, <, \leq\}$. As a convention, we use quoted strings to denote literals (e.g., “Julie”, “4”). Natural numbers are denoted by English letters towards the beginning of the alphabet such as $a$ and $b$ whereas RDF terms are denoted by the letters $s$, $p$, and $o$ or using English letters towards the end of the alphabet such as $u$ and $v$. For values that can be either natural numbers or RDF terms, we use Greek letters such as $\alpha$ and $\beta$.

3.1.2 Syntax

Definition 3.1. The set of all RDF.co expressions $E$ is defined inductively as follows:

1. **Atomic**: if $g$ is a graph symbol (i.e., $g \in \mathcal{C}$) then $g$ is an RDF.co expression.

2. **Other RDF.co Expressions** are defined using one of the following operators:

   (a) **Accessing Binding** ($bAccess$): if $e$ is an RDF.co expression then $(e.b)$ is also an RDF.co expression.

   (b) **Projection Expressions** ($\text{PROJ}$): if $e$ is an RDF.co expression and $(a_1, ..., a_n)$ is a sequence of natural numbers, then $(e|_{(a_1, ..., a_n)})$ is an RDF.co expression. For example $(e|_{(4,2)})$ is a PROJ expression.
(c) Binding Extension Expressions (bEXT): if e is an RDF.co expression, \( h \in \mathcal{H} \) is an n-ary function symbol (i.e., a function that takes n parameters) and \( a_1, ..., a_n \) is a sequence of natural numbers then \((e \oplus (a_1, ..., a_n)) h\) is also an RDF.co expression.

(d) Graph Extension Expressions (gEXT): if e is an RDF.co expression, \( \alpha_1, \alpha_2, \alpha_3 \) are three natural numbers or RDF terms (i.e., \( \alpha_1, \alpha_2, \alpha_3 \in \mathcal{T} \cup \mathbb{N} \)) then \((e \oplus (\alpha_1, \alpha_2, \alpha_3))\) is also an RDF.co expression.

(e) Triple Pattern Matching Expressions (MATCH): If e is an RDF.co expression and \( t \) is a triple pattern then \((e[t])\) is also an RDF.co expression.

(f) Filtering Expressions (FILTER): if e is an RDF.co expression, \( a, b \in \mathbb{N} \) and \( u, v \in \mathcal{T} \) then the following are valid RDF.co expressions: \((e[\theta b])\), \((e[\theta u])\) and \((e[\theta v])\) where \( \theta \in \{=, \neq, <, \leq\} \).

(g) Cross Product Expressions (CROSS): if \( e_1 \) and \( e_2 \) are RDF.co expressions, then so is \((e_1 \times e_2)\).

(h) No other elements are in \( \mathbb{E} \).

### 3.1.3 Semantics

URIs and literals are interpreted as themselves (i.e., a Herbrand interpretation). We define an RDF triple\(^3\) as a triple \((s, p, o) \in \mathcal{U} \times \mathcal{U} \times \mathcal{T}\). In this triple, \( s \) is the subject, \( p \) is the predicate and \( o \) is the object. An RDF graph is a set of RDF triples. We denote the set of all RDF graphs over a set of RDF terms \( \mathcal{T} \) as \( \mathcal{G} \) (i.e., \( \mathcal{G} = 2^\mathcal{T} \)). We also use \( g^\phi \) to refer to the empty RDF graph (i.e., \( g^\phi = \emptyset \)).

A binding is a sequence of RDF terms (URIs or literals). The length of a binding \( S \) is denoted as \( |S| \) and the empty binding is denoted as \( S^\phi \) (notice that \( |S^\phi| = 0 \)). We use subscript to access binding elements based on their positions in the sequence (i.e., the \( i \)th element of a binding \( S \) is \( S_i \)). Trying to access an element by an index that is zero or bigger than the length of a binding yields an error. We denote the set of all bindings as \( S \).

\(^2\)For simplicity of presentation, we restrict filters to a single predicates. However, extending this to add logical operators is straightforward.

\(^3\)We only consider ground RDF graphs and therefore we do not consider blank nodes.

\(^4\)We use \( 2^\mathcal{T} \) to denote the power set of a set \( \mathcal{T} \).
The concatenation of two bindings \( S = (u_1, u_2, \ldots, u_n) \) and \( T = (v_1, v_2, \ldots, v_m) \) is the binding \((u_1, u_2, \ldots, u_n, v_1, v_2, \ldots, v_m)\). We denote concatenation by a dot (i.e., \( S \cdot T \)).

We now define the semantics of RDF.co expressions. Let \( M : C \rightarrow G \) be an assignment function that maps each symbol from \( C \) (i.e., an atomic expression) to an RDF graph. For each expression \( e \) the value (or interpretation) of it under graph symbol assignment \( M \) is denoted as \([e]_M\). The value is always a set of pairs of a graph and a binding. In other words, the interpretation function \([\cdot]_M \) assigns to each valid RDF.co expression a value from \( 2^{G \times S} \) (i.e., \([\cdot]_M : E \rightarrow 2^{G \times S})\). We capture the semantics of RDF.co expressions in the next definition and provide a number of illustrating examples afterwards.

**Definition 3.2.** The semantics of RDF.co expressions given a graph assignment function \( M \) is defined inductively as follows:

1. **Atomic:** \([g]_M = \{(M(g), S^o)\}\)

2. **bAccess:** \([ (e.b) ]_M = \{(g^\phi, S) | (g, S) \in [e]_M\}\)

3. **PROJ:** \([ (e|_{a_1,\ldots,a_n}) ]_M = \{(g, S') | (g, S') \in [e]_M, S = (S'_a_1, \ldots, S'_a_n)\}\)

4. **bEXT:**

\[
\left[ (e \oplus_{a_1,\ldots,a_n} h) \right]_M = \{(g, S \cdot (h(S_{a_1}, \ldots, S_{a_n}))) | (g, S) \in [e]_M\}
\]

The function symbol \( h \) is interpreted as a function that takes \( n \) RDF terms and produces an RDF term (i.e., \( h : T^n \rightarrow T \)).

5. **gEXT:** We define a function \( \text{value} : (T \cup N) \times S \rightarrow T \) as follows:

\[
\text{value}(\alpha, S) = \begin{cases} 
S_\alpha & \text{if } \alpha \in N \\
\alpha & \text{if } \alpha \in T
\end{cases}
\]

Notice that \( T \) and \( N \) are disjoint and therefore there is no ambiguity in the previous function definition.

\[
\left[ (e \oplus (\alpha_1, \alpha_2, \alpha_3)) \right]_M = \{(g \cup \{\text{value}(\alpha_1, S), \text{value}(\alpha_2, S), \text{value}(\alpha_3, S)\}, S) | (g, S) \in [e]_M\}
\]

6. **MATCH:**

We discuss each possible triple pattern separately assuming \( s, p, o \in T \)
Chapter 3. **RDF.co**

7. **FILTER:**

For interpreting comparison operators between RDF terms, we use the same interpretation of SPARQL as defined in section 17.3 of the W3C specification [72].

\[
\text{[[e(s,p,o)]]}_M = \{( (g, S) \mid (g, S) \in [e]_M \wedge (s, p, o) \in g\} \\
\text{[[e(s,p,:)]]}_M = \{( (g, S) \mid (g, S) \in [e]_M \wedge (s, p, o) \in g\} \\
\text{[[e(s,?,o)]]}_M = \{( (g, S) \mid (g, S) \in [e]_M \wedge (s, p, o) \in g\} \\
\text{[[e(s,?,?)]]}_M = \{( (g, S) \mid (g, S) \in [e]_M \wedge (s, p, o) \in g\} \\
\text{[[e(?p,o)]]}_M = \{( (g, S) \mid (g, S) \in [e]_M \wedge (s, p, o) \in g\} \\
\text{[[e(?p,?)]]}_M = \{( (g, S) \mid (g, S) \in [e]_M \wedge (s, p, o) \in g\} \\
\text{[[e(??,?)]]}_M = \{( (g, S) \mid (g, S) \in [e]_M \wedge (s, p, o) \in g\} \\
\text{[[e(??,?)]]}_M = \{( (g, S) \mid (g, S) \in [e]_M \wedge (s, p, o) \in g\} \\
\]

8. **CROSS:**

\[
\text{[[e1 \times e2]]}_M = \{( (g_1 \cup g_2, S, T) \mid (g_1, S) \in [e_1]_M \wedge (g_2, T) \in [e_2]_M\} \\
\]

Next, we provide illustration and examples of the semantics of RDF.co expressions.

To depict the values in this chapter, we denote each pair by drawing a graph and a table close to each other. The table represents a binding and uses the order of elements in the binding as columns headers. The set of pairs, that constitutes an expression value, is surrounded by curly brackets. In the figures, sub-figure (a) is the input while sub-figure (b) shows the result of applying an operator to the input (see Figure 3.1 for an example).

1. **Atomic:** The value of an atomic expression is a set containing a single pair.

The graph component is interpreted according to the graph symbols assignment $M$ while the binding component is the empty binding.

2. **bAccess:** **bAccess** empties the graph component but keeps the binding component of each item in the original expression value. Figure 3.1 shows an example evaluation of **bAccess**.
3. **PROJ**: A PROJ expression allows choosing a sub-sequence of the binding while leaving the graph component in each pair unaffected. Figure 3.2 provides an example of a projection expression value.

![Figure 3.2: PROJ example](image)

4. **bEXT**: These expressions allow extending the binding with a new value that is calculated based on existing values in the binding. See Figure 3.3 for an example.

Notice that in **bEXT** expressions, $h$ can be viewed as a Skolem function arising from the quantification $\forall S_a \forall S_{a2} \ldots \forall S_{an} \exists c : c = h(S_{a1}, \ldots, S_{an})$.

5. **gEXT**: A **gEXT** expression allows adding a new triple to a graph. These expressions are similar to **bEXT** expressions but they extend the graph instead of extending the binding. An example evaluation of such expression can be seen in Figure 3.4.
6. **MATCH**: MATCH expressions filter graphs to only triples matching the provided pattern and introduces the corresponding bindings. A key difference from SPARQL pattern evaluation is retaining the matching triples in addition to the bindings. Figure 3.5 shows an example. Notice that a triple pattern matching expression yields a graph with only one triple and eliminates previous bindings. Notice also that one can still apply further pattern matching on the results, something that is not possible in SPARQL.

7. **FILTER**: When natural numbers are used in the filter expression, they are interpreted as indices used to access items in the binding. Values that are URIs (U) or literals (L) are interpreted into themselves. Notice that both U and L are disjoint from N and therefore there is no ambiguity in interpreting a FILTER expression.
8. **CROSS**: A CROSS expression unites the graph components and concatenates the binding components. See Figure 3.6 for an example.

It is worth mentioning that the defined operators can express select-project-join queries (aka conjunctive queries). They are similar to conjunctive queries that formed the basis of earlier studies of the relational data model. Those queries have unique characteristics such as monotonicity and satisfiability [5]. Further studies on the relational model extended the set of operators considered and studied further operators such as negation and recursion. For RDF.co, we define operators required to express select-project-join queries and leave studying further operators for future work.
3.2 Properties of RDF.co Expressions

Remark 3.1. We omit parentheses when writing RDF.co expressions when this can be done without ambiguity. In particular, outermost parentheses can always be omitted. Furthermore, we will drop $M$ and write $[.]$ instead of $[.]_M$ when no ambiguity results from this omission.

Theorem 3.2. Let $e$ be an RDF.co expression, then there exists an integer $n \geq 0$ such that for all $(g, S) \in [e], |S| = n$.

Proof. We prove the theorem by structural induction.

Basis. if $e$ is an Atomic expression then the theorem is correct by taking $n = 0$ (i.e., $[e]$ is a singleton of a pair that has $S^\emptyset$ as the binding component and $|S^\emptyset| = 0$).

Induction. We assume that the theorem is true for all expressions $e'$ that are used in the recursive definition of the expression $e$ and then prove the theorem for $e$. We discuss each of the operators in Definition 3.1.

- if $e$ is a PROJ expression of the form $e'|_{(a_1,a_2,...,a_n)}$ then for all $(g, S) \in [e]$ it is the case that $|S| = n$.

- if $e$ is bEXT expression $e' \oplus_{(a_1,...,a_m)} h$, then for all $(g, S) \in [e]$ it is the case that there exists some $(g', S') \in [e']$ such that $|S| = |S'| + 1$. But given the induction hypothesis, there exists some integer $n \geq 0$ such that $|S'| = n$ for all $(g', S') \in [e']$. Consequently, $|S| = n + 1$ for all $(g, S) \in [e]$.

- if $e$ is a MATCH expression of the form $e'[t]$, then for all $(g, S) \in [e], |S| =$ number of $?$ in $t$. The theorem is proved by setting $n$ to the number of $?$ in $t$.

- if $e$ is a CROSS expression of the form $e_1 \times e_2$ then, based on the induction hypothesis, there exists two numbers $n_1$ and $n_2$ such that $\forall (g_1, S_1) \in [e_1] \forall (g_2, S_2) \in [e_2] : |S_1| = n_1 \land |S_2| = n_2$. However, based on the definition of CROSS expressions, for all $(g, S) \in [e]$ it is the case that $S = S_1 . S_2$ for some $(g_1, S_1) \in [e_1]$ and $(g_2, S_2) \in [e_2]$. Hence, $|S| = |S_1| + |S_2| = n_1 + n_2$ (i.e., $|S|$ is a constant for all $(g, S) \in [e]$).
• if \( e = e'.b \), \( e = e'[x \theta y] \), or \( e = e' \oplus (\alpha_1, \alpha_2, \alpha_3) \) (i.e., \( e \) is a \texttt{bAccess}, \texttt{FILTER}, or \texttt{gEXT} expression), then every binding of an element in \([e]\) has the same length of a binding of some element in \([e']\). Hence, given the induction hypothesis, the binding of all elements in \([e']\) has the same length \( n \) and, therefore, it is also the case for items in \([e]\).

Accordingly, the induction holds for all \texttt{RDF.co} expressions. Therefore, by the principle of induction, Theorem 3.2 holds for all \texttt{RDF.co} expressions.

The previous theorem allows us to refer to the length of binding of an expression \( e \). We denote this as \(|e|_b\). Furthermore, based on Theorem 3.2, it can be inferred that when applying an operator to some expression \( e \), the operator either yields no error or yields an error on each item of \([e]\). This can be inferred from Theorem 3.2 by noticing that errors can occur when trying to access bindings using indices that are zero or larger than the binding length. Additionally, the same length property of bindings can be utilised to optimise computation and storage.

The next theorem states the primitivity of the \texttt{RDF.co} operators.\(^5\)

**Theorem 3.3.** \texttt{RDF.co} operators are primitive (i.e., none of them can be defined in terms of the other operators).

**Proof.** We show that each operator has a unique characteristic that no other operator has. To prove a unique characteristic of an operator \( \rho \), we drop \( \rho \) from \texttt{RDF.co} and show that: using the new smaller set of operators, some results, that were possible before, cannot be achieved anymore. For some operators, their unique characteristic is clear and therefore is stated directly. For some other operators, we show their uniqueness via an example. Figure 3.7 provides a pictorial illustration that might help clarifying the proof.

1. \texttt{bAccess}: it is the only operator that can drop the graph component of an expression value without changing its binding component.

2. \texttt{PROJ}: it is the only operator that can reduce the length of a binding.

\(^5\)The proof is inspired by the proof of primitivity of relational algebra operators presented in a lecture by Prof. Phokion G. Kolaitis. The lecture is available at https://users.soe.ucsc.edu/~kolaitis/talks/gii09-final.pdf.
3. \texttt{bEXT}: it is the only operator that can introduce a new value in the binding. Notice that while \texttt{MATCH} operators manipulate the bindings, they do not introduce a value that did not exist in the graphs of the involved expressions.

For instance, an expression \( e' \) with a value \( \llbracket e' \rrbracket = \{ (\{\},(1)),(\{\},(2)) \} \), can be achieved with an \texttt{RDF.co} expression without \texttt{bEXT}. Nevertheless, starting from \( e' \), getting an expression \( e \) with a value of \( \llbracket e \rrbracket = \{ (\{\},(1,3)),(\{\},(2,3)) \} \) cannot be achieved without using \texttt{bEXT}. While using \texttt{gEXT} can introduce the value 3 to the graph components, transferring this value to the binding component using a \texttt{MATCH} operator replaces existing binding and eliminates the values 1 and 2 from the binding components. See Figure 3.7 for an illustration.

Notice that the intended value can be achieved using \texttt{bEXT} via \( e' \oplus \{\} h_3 \) where \( h_3 \) is a 0-ary function that produces the value 3. Notice also that \( e' \) can be expressed using \texttt{RDF.co} without \texttt{bEXT} as follows: \( e' = (g[(s,p,?)]).b \) with \( M(g) = \{ (s,p,1),(s,p,2) \} \).

4. \texttt{gEXT}: it is the only operator that can introduce new values to the graph components in an expression value.

5. \texttt{MATCH}: it is the only operator that can shrink the size of graph components.

6. \texttt{FILTER}: it is the only operator that can reduce the size of an expression value even when all elements have equivalent graph components.

For instance, an expression \( e' \) with a value
\[
\llbracket e' \rrbracket = \{ ((s,p,1),(s,p,2)),(1))),( (s,p,1),(s,p,2), (2)) \}
\]
can be achieved with \texttt{RDF.co} without \texttt{FILTER}. Nevertheless, starting from \( e' \), getting an expression \( e \) with a value of \( \llbracket e \rrbracket = \{ ((s,p,1),(s,p,2)),(1)) \} \) cannot be achieved without using \texttt{FILTER}. One can try to use \texttt{MATCH} operator to reduce the size of the expression value. However, because the two elements in the expression value have the same graph component, \texttt{MATCH} can only affect both of them or neither. On the other hand, \texttt{PROJ} operator can be tried to reduce the size of the expression value. However, \texttt{PROJ} operator reduces the size only when binding components of two items become identical. This cannot be achieved as the bindings in the value of \( e' \) are different. All other operators cannot reduce the size of the expression value.
Notice that the intended value can be achieved using `FILTER` via \( e[1 = \text{"1"}] \).
Notice also that \( e' \) can be expressed in `RDF.co` without `FILTER` as follows: \( e' = g[(s, p, ?)] \times g \) with \( M(g) = \{(s, p, 1), (s, p, 2)\} \). See Figure 3.8 for pictorial representation of \( e' \).

7. **CROSS**: it is the only operator that can bring the values of two `RDF.co` expressions together. `CROSS` operator unions graphs and concatenates bindings in the two expressions values. As a value of an expression can only be known when that expression is interpreted, `CROSS` operators cannot be replaced by a sequence of `bEXT` and `gEXT` operators.

We showed that each of the `RDF.co` operators has a unique characteristic that other operators do not have. Therefore, any of the `RDF.co` operators cannot be defined using the other operators.

![Figure 3.7: Illustrative representation of operators that transfer values between an expression (left) and the result of applying an operator to it (right)](image)

Theorem 3.3 shows that the set of `RDF.co` operators as defined in Definition 3.1 is minimal. Next, we use this minimal set to define a number of derived operators.
3.3 Derived Expressions

In this section, we list a number of derived operators that can be defined using RDF.co operators. Having these operators provides convenience and brevity to the data model but does not alter neither the expressivity nor the computational complexity of the data model. Therefore, these operators can be considered as abbreviations to other equivalent and more verbose expressions. First we define the notion of equivalence between two RDF.co expressions.

**Definition 3.3.** Two RDF.co expressions $e_1$ and $e_2$ are said to be equivalent when for any graph symbol assignment function $M$, it is the case that $(g,S) \in q_{e_1} \iff (g,S) \in q_{e_2}$ $y_M$. We denote equivalence by writing $e_1 \equiv e_2$.

### 3.3.1 Graph Access (gAccess)

**Syntax:** If $e$ is an RDF.co expression, the so is $e.g$.

**Semantics:** $\llbracket (e.g) \rrbracket = \{(g,S) \mid (g,S) \in \llbracket e \rrbracket \}$

**gAccess** keeps the graph component and empties the binding component of each item in the original expression value. Notice that $e.g \equiv e|_{(0)}$ (i.e., a projection with empty list of columns yields the graph components only). Figure 3.9 provides an example of gAccess.
3.3.2 Equi Join (E-JOIN)

**Syntax:** If $e_1$ and $e_2$ are RDF.co expressions, $i$ and $j$ are two natural numbers then $(e_1 \bowtie_{i=j} e_2)$ is an RDF.co expression.

**Semantics:** $\sem{(e_1 \bowtie_{i=j} e_2)} = \{ (g_1 \cup g_2, S \cdot (T_1, \ldots, T_{j-1}, T_{j+1}, \ldots, T_m) ) \mid |e_2|_b = m \land (g_1, S) \in \sem{e_1} \land (g_2, T) \in \sem{e_2} \land S_i = T_j \}$

E-JOIN is equivalent to the expression: $\sem{(e_1 \times e_2)[i = j + n]}(1, \ldots, n, n+1, \ldots, n+j-1, n+j+1, \ldots, n+m)$

Where $|e_1|_b = n$ and $|e_2|_b = m^6$.

Figure 3.10 shows an example of E-JOIN operator and a step-by-step evaluation of its equivalent expression. It is worth mentioning that this operator resembles relational algebra equi-join.

3.3.3 Graph Pattern Matching (GMATCH)

A graph pattern is a shortcut to apply multiple triple patterns and “join” the results. A GMATCH expression uses a sequence of triple patterns along with a set of equality conditions of the form $i = j$ where both $i$ and $j$ are natural numbers referring to positions of terms in the triple patterns.

\footnote{We remind the reader that $|e|_b$ is the notion of length of binding introduced as a result of Theorem 3.2.}
Syntax: If \( e \) is an RDF.co expression, \( t_1, t_2, \ldots, t_n \) are triple patterns and \( i_1, \ldots, i_m, j_1, \ldots, j_m \) are natural numbers then \( (e[(t_1, t_2, \ldots, t_n), i_1 = j_1, \ldots, i_m = j_m]) \) is an RDF.co expression.

Semantics: A GMATCH expression matches each of the triple patterns \( t_1, \ldots, t_n \) separately; calculates the cross product of the results; applies the filters given by the predicates \( i_k = j_k \); then projects out one of the equivalent values in the binding similar to the way E-JOIN expressions do. We provide the formal definition of the semantics of GMATCH expressions by stating their equivalent expressions:

\[
e[(t_1, t_2, \ldots, t_n), i_1 = j_1, \ldots, i_m = j_m] \equiv (e[t_1] \times e[t_2] \ldots \times e[t_n])[i_1 = j_1][i_m = j_m][(1 \ldots k) \setminus \{j_1, \ldots, j_m\}]
\]

Where \( (1 \ldots k) \setminus \{j_1, \ldots, j_m\} \) is the sequence of numbers from 1 to \( k \) without the numbers in the set \( \{j_1, \ldots, j_m\} \) and \( k \) is the length of binding of the expression \( (e[t_1] \times e[t_2] \ldots \times e[t_n]) \).
For example, $e[ ((s, ?, ?), (p, ?), (? , o)), 2 = 3, 4 = 5 ]$ is equivalent to $(e[(s, ?, ?)] \times e[(?, p, ?)] \times e[(?, o)])[2 = 3][4 = 5](1,2,4,6)$

Figure 3.11 shows an example of a graph pattern expression and its equivalent form. GMATCH expressions have a direct resemblance to basic graph pattern matching of SPARQL. We will come back to this resemblance when discussing the relationship between RDF.co expressions and SPARQL in Section 3.4.

### 3.3.4 Multiple Graph Extensions (mgEXT)

mgEXT expressions allow adding multiple triples at once. We first show that the order of adding multiple triples does not matter and then use this result to define mgEXT.

**Lemma 3.4.** If $e$ is an RDF.co Expression and $\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \beta_3$ are natural numbers or RDF terms (i.e., $\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \beta_3 \in T \cup \mathbb{N}$) then $(e \oplus (\alpha_1, \alpha_2, \alpha_3)) \oplus (\beta_1, \beta_2, \beta_3) \equiv (e \oplus (\beta_1, \beta_2, \beta_3)) \oplus (\alpha_1, \alpha_2, \alpha_3)$

**Proof.** Applying gEXT operator to some expression $e$ does not change the binding component of the value of $e$ but only adds a new triple to the graph component. As described before, the evaluation of a gEXT operator uses a function $value : (T \cup \mathbb{N}) \times S \rightarrow T$ that is defined as follows:

$$value(\alpha, S) = \begin{cases} S_\alpha & \text{if } \alpha \in \mathbb{N} \\ \alpha & \text{if } \alpha \in T \end{cases}$$

Therefore, the components of newly added triples that the function $value$ introduces are either constant RDF terms or terms in the binding accessed by their indices. As bindings do not change when applying gEXT, the order of applying it multiple times does not change the results. Hence, $(e \oplus (\alpha_1, \alpha_2, \alpha_3)) \oplus (\beta_1, \beta_2, \beta_3) \equiv (e \oplus (\beta_1, \beta_2, \beta_3)) \oplus (\alpha_1, \alpha_2, \alpha_3)$

As a result of Lemma 3.4, we will use a shortcut and write $e \oplus \{(a_1, a_2, a_3), (b_1, b_2, b_3)\}$ instead of $(e \oplus (a_1, a_2, a_3)) \oplus (b_1, b_2, b_3)$. We will refer to this as a multiple graph extension operator mgEXT.
Figure 3.11: Evaluation of the GMATCH expression: $e[([?,p1,?]) \times e([?,p2,?])][2 = 3] = e[([?,p1,?]),([?,p2,?]),2 = 3]$
3.4 Relationship to SPARQL

SPARQL is the language recommended by the W3C to query RDF data. The syntax and semantics of SPARQL is formally defined in its specification [72]. SPARQL defines different types of queries (i.e., SELECT, CONSTRUCT, ASK, and DESCRIBE). In this section, we compare the expressivity of the RDF.co expressions to that of SPARQL and discuss a number of differences between the two models.

3.4.1 RDF Datasets in SPARQL

A SPARQL query is executed against an RDF dataset which represents a collection of graphs. An RDF dataset comprises one graph, the default graph, which does not have a name, and zero or more named graphs, where each named graph is identified by an IRI [72]. In other words, a dataset of a SPARQL query is a set \( \{G, (u_1, G_1), \ldots, (u_n, G_n)\} \) where \( G \) and each \( G_i \) are RDF graphs, and each \( u_i \) is a distinct IRI. \( G \) is a called the default graph. \( G_i \) are named graphs. Moreover, using GRAPH keyword in SPARQL allows a pattern to be matched against a particular named graph. In SPARQL Algebra, if \( P \) is a pattern and \( u \) is an IRI, then \( u \) GRAPH \( P \) is also a pattern [72].

In RDF.co the graph symbol set \( C \) together with the mapping \( M \) under which an expression is interpreted can provide equivalent notion of RDF datasets. They also allow evaluating an expression against a particular graph. However, in comparison to RDF datasets in SPARQL, graph symbols in RDF.co are not limited to the set of IRIs and they are valid expressions on their own. Whereas named graphs are not valid SPARQL expressions. In fact, a dataset definition is part of the SPARQL query definition. For example, In [9, 134], a SPARQL query is defined as a quadruple \( (V, P, DS, SM) \) where \( V \) is a result form (i.e., SELECT, CONSTRUCT, ASK, or DESCRIBE), \( P \) is a pattern, \( DS \) is a dataset, and \( SM \) is solution modifiers (e.g., distinct, order, etc.). In RDF.co, a dataset is not part of the query definition.

On the other hand, one limitation of RDF.co is that graph names cannot be variables. In SPARQL algebra, \(?x\) GRAPH \( P \) where \(?x\) is a variable is a valid pattern. A variable graph name causes a pattern to be matched against all named graphs in the dataset. RDF.co has no equivalent notion.
3.4.2 Expressing SPARQL SELECT Queries Using RDF.co Expressions

**Theorem 3.5.** RDF.co expressions can define SPARQL 1.1 SELECT queries with basic graph patterns, filters, and assignments.

**Proof.** In SPARQL, a basic graph pattern is a set of triple patterns joined together based on shared variable names. GMATCH, which was defined as a derived RDF.co operator, comprises a set of triple patterns where joins between different patterns are defined by variable positions. Therefore, a SPARQL basic graph pattern can be expressed using GMATCH by introducing any order over the triple patterns and replacing variable names by their positions. SPARQL filters can be defined using FILTER expressions and SPARQL assignments with bEXT expressions.

The bAccess operator allows ignoring the graph component of the results to get only a table similar to SPARQL results, whereas the SELECT clause of a SPARQL query can be defined using PROJ expression.

We next show a couple of example SPARQL queries along with their equivalent RDF.co expressions:

- The SPARQL query: `SELECT ?s ?v WHERE { ?s :p ?o . ?o :p2 ?v }` evaluated on the RDF graph $g$ can be expressed using the expression: $(g[(?,?),(?,:p2,?)],2=3).b_{(1,3)}$
  Alternatively, the derived operator GMATCH can be expanded to its equivalent form using the main RDF.co expressions only: $(((g[(?,?)] \times g[(?,?)])[2=3]).b)_{(1,4)}$

- The SPARQL query: `SELECT ?s ?z WHERE { ?s :p ?o . ?o :p2 ?v BIND(?v * 1.1) AS ?z }` evaluated on the RDF graph $g$ is equivalent to the expression: $(g[(?,?),(?,:p2,?)],2=3) \odot_{(3)} \times_{1,1}.b)_{(1,4)}$
Where $x_{1.1}(x) = x \times 1.1$ (i.e., $x_{1.1}$ is a function that multiplies its parameter by 1.1).

Similarly, by expanding \texttt{GMATCH} to its equivalent form using the main \texttt{RDF.co} expressions only:

$$(((g([(?, :p, ?]) \times g([(?, :p2, ?)])[2 = 3])) \oplus (4 \times 1.1 \cdot b))_{(1,5)}$$

### 3.4.3 Expressing SPARQL CONSTRUCT Queries Using \texttt{RDF.co} Expressions

**Theorem 3.6.** \texttt{RDF.co} expressions can define SPARQL 1.1 CONSTRUCT queries with basic graph patterns, filters and assignments as long as no blank nodes are used in the CONSTRUCT query template.

**Proof.** Similar to the proof of Theorem 3.5 we can show that \texttt{RDF.co} expressions can build the binding table corresponding to the body of a SPARQL query that uses basic graph patterns with filters and assignments. The CONSTRUCT template that is applied on top of the binding table to get the result graph is a set of triples that uses direct RDF terms (constants) and variables bound by the query body. This template can be defined using \texttt{mgEXT} to get equivalent graph using \texttt{RDF.co}.

However, \texttt{mgEXT} cannot express blank nodes in CONSTRUCT templates. In SPARQL SELECT queries, blank nodes can be replaced with fresh query variables. Whereas a blank node appearing in a query’s CONSTRUCT clause is replaced by a fresh blank node for each solution mapping in the resulting RDF [115]. \texttt{RDF.co} does not have a notion of blank nodes.

As an example of a CONSTRUCT query, The following query:

CONSTRUCT { ?s :p ?o . ?o :newVal ?z . }
WHERE { ?s :p ?o . ?o :p2 ?v BIND(?v * 1.1) AS ?z }

evaluated on the RDF graph $g$ is equivalent to the expression:
(g[([?, p, ?], [?, p2, ?]), 2 = 3] ⊕_{1.1} b ⊕ (1, p, 2), (2, :newVal, 4))

Where \( x_{1.1}(x) = x \times 1.1 \) (i.e., \( x_{1.1} \) is a function that multiplies its parameter by 1.1).

Similarly, by expanding both \texttt{GMATCH} and \texttt{mgEXT} to their equivalent forms using the main \texttt{RDF.co} expressions only:

(((g[([?, p, ?])] \times (g[([?, p2, ?])])[2 = 3]).b \oplus (4) \times_{1.1} \oplus (1, p, 2)) \oplus (2, :newVal, 5))

### 3.4.4 Discussion

The previous sections show that \texttt{RDF.co}, as defined in this chapter, provides a uniform definition that is able to provide a consistent formulation of SPARQL datasets, named graphs, SELECT and CONSTRUCT queries. However, the main advantage and motivation of \texttt{RDF.co} is its ability to arbitrarily cascade expressions. In SPARQL, it is not possible to “join” or “union” two graphs. It is also not possible to cascade pattern matching (i.e., it is not possible to directly apply pattern matching against the result of another pattern matching). Such restrictions complicate nesting SPARQL queries and impose extra operators to obtain the required results as shown in the next two example SPARQL queries:

- **CONSTRUCT** \{?s ?p ?o\} \texttt{WHERE} \{?s ?p ?o\} which is commonly used to get the total graph behind some end point. One cannot ask directly for the graph.

- **SELECT** ?s ?p ?o \texttt{WHERE} \texttt{GRAPH<http://graph1IRI>} \{?s ?p ?o\} \texttt{UNION} \texttt{GRAPH<http://graph2IRI>} \{?s ?p ?o\} to union two graphs. One cannot directly union the two graphs.

### 3.4.4.1 The Set of Operators in \texttt{RDF.co}

The operators included in \texttt{RDF.co} allow expressing select-project-join queries and allows defining new values via extending bindings and graphs. Furthermore,
RDF.co adopts graph pattern matching similar to SPARQL. It is worth mentioning that in databases, support for select-project-join is sufficient to call a system “minimally relational” [40]. However, RDF.co misses many other important operators, such as optional matching and union operators. The set of operators was chosen mainly based on simplicity of both definition and distributed implementation. It has been a common practice for big data languages to include a small set of operators amenable to parallelisation and allow extension via user-defined functions. There is still certainly a significant value in adding further operators to RDF.co, particularly for optional matching. This thesis, nevertheless, does not address this need.

3.4.4.2 Nesting Queries in SPARQL

SPARQL 1.1 supports subqueries by allowing a SELECT query to be used in the place of a graph pattern. The result of the subquery is joined with other patterns in the query. Only variables projected out of the subquery (i.e., included in the SELECT clause) are joined; in order to be compatible with the original semantics of SPARQL graph patterns [9]. In RDF.co, this type of nesting can easily be achieved by defining the nested subquery as an expression and joining it with the outer query.

Nesting can also be achieved in SPARQL by using a CONSTRUCT query in the FROM clause. Nesting CONSTRUCT queries was proposed in [144] but was not adopted in the standard SPARQL 1.1 language. Recently, it was shown that FROM CONSTRUCT nesting can be achieved in SPARQL 1.1 using the VALUES expressions [135]. Such nesting can be achieved in RDF.co as expressions can be cascaded arbitrarily and they all have both a graph and a binding component. In other words, RDF.co does not distinguish between CONSTRUCT and SELECT queries.

Another form of nesting SPARQL queries is by using ASK queries in FILTER clauses as proposed in [134] and adopted in SPARQL 1.1 via the filter operator EXISTS. This cannot be achieved in RDF.co. All RDF.co operators manipulate a single item in an expression value. In other words, there is no operator in RDF.co that has a global view of the value and can tell, for instance, whether a given expression has an empty value. Adding such operator allows RDF.co to express SPARQL ASK queries and EXISTS filter operator.
3.4.4.3 RDF.co Limitations

**Missing Operators.** As stated before, RDF.co cannot express SPARQL OPTIONAL matching and set operators, in particular UNION. This results from the absence of a notion of unbound variables in RDF.co. OPTIONAL clauses in SPARQL is the main differentiation between SPARQL and traditional relational query languages. However, it adds to the complexity of SPARQL [129]. In the presence of unbound variables, multiple semantics of joins can exist and can yield different results [134].

**Blank nodes.** Blank nodes are defined in RDF as “existential variables”. In practice, treatment of blank nodes does not comply always with the intended semantics [115]. In SPARQL queries, blank nodes can be replaced by fresh variables, except in a **CONSTRUCT** query template. RDF.co works only with ground RDF graphs (i.e., graphs with no blank nodes). The local scope of blank nodes makes them challenging to support in a distributed environment. RDF molecule as defined in [50] can be a promising approach for preserving blank nodes in distributed RDF graphs. We do not discuss support for blank nodes further in this thesis.

**Bag semantics.** We only showed the description of RDF.co in set semantics (i.e., no duplicates allowed in values). This can be generalised for bag semantics. Nevertheless, care needs to be taken to redefine the operators precisely in bag semantics and vets the validity of the results that were proven based on set semantics.

3.5 Summary

In this chapter, we introduced RDF.co with a formal definition of its syntax and semantics. We discussed the expressivity of the model by comparing it to that of SPARQL. The two main distinctions of RDF.co from SPARQL are (i) its ability to cascade triple patterns matching and (ii) its use of two-component values (graph and binding components). In the next chapter we study the consequences of these differences on the characteristics of the model and the corresponding optimisation opportunities.
Chapter 4

Algebraic Properties of RDF.co Expressions

Query optimisation played a central role in the success of relational databases and was the focus of multiple research endeavours [11, 71, 148, 161]. To deal efficiently with large RDF datasets, Semantic Web researchers have also studied query optimisation [129, 146]. As a result, a number of optimisation techniques have been developed, mostly in the context of SPARQL queries. Examples include graph patterns reordering based on selectivity estimation [123, 152], normal forms [129], specialised indices [65, 73, 169], and translating SPARQL queries into SQL [35, 43] and datalog [134].

In this chapter we study query optimisation for RDF.co introduced in the previous chapter. We focus on algebraic properties of the data model. Algebraic properties, when interpreted as rewriting rules, provide theoretical foundation needed to apply cost-based query optimisation. Algebraic properties of the SPARQL algebra have been studied in the seminal work of Pérez et al. [129] and Schmidt et al. [146]. In this chapter, we focus on the two main distinctions of RDF.co from SPARQL. Properties related to the ability to cascade triple patterns matching in RDF.co are presented first (Section 4.1). Next, we discuss optimisation related to the use of two-component values (graph and binding components) in RDF.co (Section 4.2).
4.1 Cascading Triple Pattern Matching Operators

Remark 4.1. The order of applying multiple MATCH operators matters. In general, for an RDF expression $e$ and two triple patterns $t_1$ and $t_2$,

$$(e[t_1])[t_2] \neq (e[t_2])[t_1]$$

For example, $(e[(s, ?, o)][(?, p, o)])$ and $(e[(?, p, o)][(s, ?, o)])$ are not equivalent. The former outputs the matching subject $s$ while the latter outputs the matching predicate $p$.

To facilitate studying cascaded application of multiple triple patterns, we define a partial ordering relationship between triple patterns. We start by defining a relationship between RDF terms and then generalise it to triple patterns before proving that the introduced relationship defines, indeed, a partial order relationship.

Definition 4.1. $\forall x, y \in \mathcal{T} \cup \{?\} : x \leq y$ if and only if one of the following holds:

- Both $x$ and $y$ are $?$
- $x$ and $y$ are equal RDF terms (i.e., $x, y \in \mathcal{T} \land x = y$)
- $x$ is a term and $y$ is $?$ (i.e., $x \in \mathcal{T} \land y = ?$

We generalise $\leq$ to triple patterns.

Definition 4.2. For two triple patterns $(x_1, x_2, x_3)$ and $(y_1, y_2, y_3)$ we say that $(x_1, x_2, x_3) \preceq (y_1, y_2, y_3)$ if and only if $x_1 \leq y_1$, $x_2 \leq y_2$, and $x_3 \leq y_3$.

Notice that the set of triples is contained in the set of triple patterns (i.e., a triple is a triple pattern that contains no $?$). Therefore, $\preceq$ is defined on triples and triple patterns.

Lemma 4.2. The relationship $\preceq$ is a partial order relationship over $\mathcal{T} \cup \{?\}$ (i.e., $\preceq$ is antisymmetric and transitive).
Chapter 4. RDF.co

Figure 4.1: A proof tree that shows that ≤ is transitive

Proof. We prove first that ≤ is antisymmetric over $\mathcal{T} \cup \{?\}$ by showing that if $x \leq y$ and $y \leq x$ then $x = y$ for any $x, y \in \mathcal{T} \cup \{?\}$.

Based on the definition of ≤, $? \not\leq x$ for any $x \in \mathcal{T}$ (i.e., ? is never ≤ than a term). If $x \leq y$ and $y \leq x$, then $x$ and $y$ are both terms or are both ?. If they are both terms, then they must be equal based on the definition of ≤. Hence, $(x \leq y \land y \leq x) \Rightarrow x = y$. Therefore, ≤ is antisymmetric over $\mathcal{T} \cup \{?\}$.

We next show that ≤ is transitive. We need to show that if $x \leq y$ and $y \leq z$ then $x \leq z$. Figure 4.1 shows a proof tree that starts with the two possibility of $x \in \mathcal{T}$ or $x = ?$. In all the leaves of the tree it is the case that $x \leq z$. Notice that the inference that is represented in the tree follows directly from the definition of ≤.

Consequently, ≤ is antisymmetric and transitive. Therefore, ≤ defines a partial order relationship over $\mathcal{T} \cup \{?\}$.

Lemma 4.3. The relationship ≤ is a partial order relationship over triple patterns (i.e., ≤ is antisymmetric and transitive).

Proof. This follows from Lemma 4.2 as ≤ is antisymmetric and transitive over $\mathcal{T} \cup \{?\}$. Generalising ≤ over the set of triple patterns is defined by applying it on
the three components of the triple patterns. However, each component of a triple pattern is a member of $\mathcal{T} \cup \{\?\}$. Hence, $\leq$ is antisymmetric and transitive over triple patterns.

The next lemma links $\leq$ to the triple pattern matching operator ($\text{MATCH}$) defined as part of RDF.co in the previous chapter (Definition 3.2).

**Lemma 4.4.** For any RDF.co expression $e$ and any triple pattern $t$, and for any triple $\alpha$, it is the case that $(g', S') \in [e[t]] \wedge \alpha \in g'$ if and only if $\exists (g, S) \in [e] \wedge \alpha \in g \wedge \alpha \leq t$.

**Proof.** This follows from the semantic definition of MATCH operator in Section 3.1.3. The semantic definition discusses eight possibilities of a triple pattern (depending on the number and positions of $?$ in a triple pattern). In all eight cases, a triple $\alpha$ is added to the results if and only if $\alpha \leq t$.

As a result of Lemma 4.4, the semantics of MATCH operators can be made more concise using $\leq$ as follows:

$$[e[t]] = \{((\{\alpha\}, S_{\alpha,t}) \exists (g, S) \in [e] : \alpha \leq t \wedge \alpha \in g\} \quad (4.1)$$

Where $S_{\alpha,t}$ is a binding that gets the values in $\alpha$ corresponding to the positions of $?$ in $t$. For example, $S_{(s,p,o),(??,?,?)} = (s,p,o)$ and $S_{(s,p,o),(?,?,?)} = (s,o)$.

**Lemma 4.5.** For any two triple patterns $t_1, t_2$ such that $t_1 \leq t_2$ and for any RDF.co expression $e$, it is the case that if $\exists (g_1, S_1) \in [e[t_1]] \wedge \alpha \in g_1$ then $\exists (g_2, S_2) \in [e[t_2]] \wedge \alpha \in g_2$.

**Proof.** For any RDF triple $\alpha$ such that $\alpha \in g_1$ for some $(g_1, S_1) \in [e[t_1]]$, based on Lemma 4.4, it is the case that $\alpha \leq t_1$. Moreover, $\exists (g, S) \in [e] \wedge \alpha \in g$. As $t_1 \leq t_2$ and $\leq$ is transitive, then $\alpha \leq t_2$. Based on the only if part of Lemma 4.4, it can be inferred that $\exists (g_2, S_2) \in [e[t_2]] \wedge \alpha \in g_2$.

Based on the previous lemma, the defined partial ordering relationship between triple patterns ($\leq$) can be thought of as a “less permissive” relationship (i.e., $t_1 \leq t_2$ intuitively means that $t_1$ is a less permissive filter than $t_2$. Any triple that is a result of applying $t_1$ to some RDF.co expression $e$ will also be a result of applying the more permissive pattern $t_2$ to $e$).
4.1.1 Triple Pattern Elimination

Applying a less specific triple pattern does not change the resulting graph. This property is formally captured by the next lemma.

**Theorem 4.6.** If $e$ is an RDF co expression and $t_1$, $t_2$ are two triple patterns such that $t_1 \preceq t_2$, then $\left( (e[t_1])[t_2]) .g \equiv (e[t_1]).g$.

**Proof.** We first show that $(g_1, S) \in \left( (e[t_1])[t_2]) .g \implies (g_1, S) \in \left( (e[t_1]).g$.

$\forall (g_1, S) \in \left( (e[t_1])[t_2]) .g \implies \exists (g_1, S) \in \left( (e[t_1])[t_2]) .g \wedge g_1 = \{\alpha\}$ for some triple $\alpha$ (based on the definitions of gAccess and MATCH. Furthermore, as $g_1$ is in the result of a MATCH operator, it contains a single triple)

$\implies \exists (g_2, S_2) \in \left( e[t_1]) \wedge g_2 = \{\alpha\} \wedge \alpha \preceq t_2$ (using Lemma 4.4)

Therefore, $g_2 = g_1 = \{\alpha\}$.

$\implies \exists (g_1, S) \in \left( e[t_1]$ (substituting $g_1$ for $g_2$)

$\implies \exists (g_1, S) \in \left( (e[t_1]).g$ (definition of gAccess).

We then need to show that $(g_1, S) \in \left( (e[t_1]).g \implies (g_1, S) \in \left( ((e[t_1])[t_2]) .g$.

$\forall (g_1, S) \in \left( (e[t_1]).g \implies \exists (g_1, S) \in \left( e[t_1]) \wedge g_1 = \{\alpha\}$ for some triple $\alpha \preceq t_1$ (using the definitions of gAccess and MATCH)

As $t_1 \preceq t_2$ and $\preceq$ is transitive, we can infer that $\alpha \preceq t_2$

$\implies \exists (g_2, S_2) \in \left( (e[t_1])[t_2]) \wedge g_2 = \{\alpha\}$ (because $\alpha \preceq t_2$ and using Lemma 4.4)

$\implies \exists (g_2, S) \in \left( ((e[t_1])[t_2]) .g \wedge g_2 = \{\alpha\}$ (definition of gAccess).

But $g_2 = g_1 = \{\alpha\}$ and therefore $(g_1, S) \in \left( ((e[t_1])[t_2]) .g$.

Hence, $\left( (e[t_1])[t_2]) .g \equiv (e[t_1]).g$.

$\square$
Based on the previous theorem, an expression can be re-written to eliminate triple patterns that do not change the results. Consequently, the cost and complexity of an expression can be reduced. As an example of applying Triple Pattern Elimination,

\[
((e[?,p,o])[(?,?,o)]).g \equiv (e[?,p,o]).g
\]

### 4.1.2 Triple Pattern Insertion

We first provide a remark that allows simplifying the proofs of the main theorems in this section and the next one.

**Corollary 4.7.** If \( e_1 \) and \( e_2 \) are two RDF.co expressions and \( t \) is a triple pattern, to show equivalence of the two expressions \( e_1[t] \) and \( e_2[t] \) it is sufficient to show that a triple \( \alpha \) is in a graph \( g_1 \) such that \((g_1, S_1) \in q_{e_1[t]}\) if and only if \( \alpha \) is in a graph \( g_2 \) such that \((g_2, S_2) \in q_{e_2[t]}\).

**Proof.** Based on Definition 4.1 of MATCH expression, \( \forall (g, S) \in [e[t]] \), then \( g \) contains a single triple \( \alpha \) and the value of \( S \) depends on \( \alpha \) and \( t \). Therefore, if a triple \( \alpha \) is in some graph in \( [e_1[t]] \) if and only if it is in some graph in \( [e_2[t]] \), then \( e_1[t] \) and \( e_2[t] \) have the same value and are, therefore, equivalent. \( \square \)

The previous remark allows simplifying testing equivalence between two RDF.co expressions that are both triple pattern matching using the same pattern. Instead of showing the equivalence per graph and binding pair, it is sufficient to show equivalence per triple.

**Theorem 4.8.** If \( e \) is an RDF.co expression and \( t_1, t_2 \) are two triple patterns such that \( t_1 \preceq t_2 \), then \( e[t_1] \equiv (e[t_2])[t_1] \).

**Proof.** We will make use of Corollary 4.7 in this proof. Firstly, for the if part, we show that if a triple \( \alpha \) is in some \( g_1 \) such that \((g_1, S_1) \in [e[t_1]]\), then \( \exists (g_2, S_2) \in [(e[t_2])[t_1]] \) such that \( \alpha \in g_2 \).

\[
\alpha \in g_1 \land (g_1, S_1) \in [e[t_1]] \implies \exists (g_2, S_2) \in [e] \land \alpha \in g_2 \land \alpha \preceq t_1 \quad (\text{using Lemma 4.4})
\]

As \( \alpha \preceq t_1 \), and \( \preceq \) is transitive, we can infer that \( \alpha \preceq t_2 \). Therefore, \( \exists (g_3, S_3) \in [e[t_2]] \land \alpha \in g_3 \) (using Lemma 4.4).
But as $\alpha \leq t_1$, applying $t_1$ to $e[t_2]$ will result in some graph that contains $\alpha$. In other words, $\exists (g_4, S_4) \in \big[ (e[t_2])[t_1] \big] \land \alpha \in g_4$.

Therefore, the if part of the theorem is proved.

Secondly, for the only if part, we show that if a triple $\alpha$ is in some $g_1$ such that $(g_1, S_1) \in \big[ (e[t_2])[t_1] \big]$, then $\exists (g_2, S_2) \in \big[ e[t_1] \big]$ such that $\alpha \in g_2$.

$$\alpha \in g_1 \land (g_1, S_1) \in \big[ (e[t_2])[t_1] \big]$$

$$\implies \exists (g_2, S_2) \in \big[ e[t_2] \big] \land \alpha \in g_2 \land \alpha \leq t_1 \quad \text{(using Lemma 4.4)}$$

$$\implies \exists (g_3, S_3) \in \big[ e \big] \land \alpha \in g_3 \land \alpha \leq t_2 \land \alpha \leq t_1 \quad \text{(using Lemma 4.4)}$$

$$\implies \exists (g_4, S_4) \in \big[ e[t_1] \big] \land \alpha \in g_4 \quad \text{(using Lemma 4.4 and the fact that $\alpha \leq t_1$).}$$

Therefore, the only if part of the theorem is proved.

Based on the previous theorem, to calculate the results of matching expression $e$ to a triple pattern $t_1$, one can instead try matching $t_1$ against the results of matching $t_2$ against $e$ (i.e., match $t_1$ against $e[t_2]$). This can be advantageous if the value of $e[t_2]$ is “cached” or “pre-computed”.

For example, it is common to partition RDF triples based on their predicates. This is known as vertical partitioning [4]. This partitioning actually can be thought of as applying the triple pattern $(?, x, ?)$ to the original data, for each predicate $x$ that is in the data, and storing each result as a partition. To match the data against a triple pattern with a bound predicate such as $(s, p, ?)$, it is sufficient to only use the partition corresponding to the predicate $p$. In other words, evaluating $(g[(?, p, ?)])(s, p, ?))$ instead of $g[(s, p, ?)]$ for each partition. Similarly, RDF data can be partitioned based on the subjects of the triples [138, 139, 168]. While such optimisations have been intuitively applied in existing systems, Theorem 4.8 provides a formal proof of the correctness of such application and allows potential use of more complex and involved expressions.
4.1.3 Triple Pattern Push-down

**Theorem 4.9.** If $e_1, e_2$ are two RDF.co expressions and $t$ is a triple pattern, then 
\[(e_1 \times e_2)[t] \equiv (e_1[t] \times e_2[t])[t].\]

*Proof.* We make use of Corollary 4.7 in this proof.

For a triple $\alpha$ such that $\alpha \in g_1$ for some $(g_1, S_1) \in q(e_1 \times e_2)[t]$ 
\[
\iff \exists (g_2, S_2) \in [e_1 \times e_2] \wedge \alpha \in g_2 \wedge \alpha \leq t \quad \text{(using Lemma 4.4)}
\]
\[
\iff ((g_2, S_2) \in [e_1] \vee (g_2, S_2) \in [e_2]) \wedge \alpha \in g_2 \wedge \alpha \leq t \quad \text{(using definition of CROSS)}
\]
\[
\iff ((g_2, S_2) \in [e_1] \wedge \alpha \in g_2 \wedge \alpha \leq t) \vee ((g_2, S_2) \in [e_2] \wedge \alpha \in g_2 \wedge \alpha \leq t) \quad \text{(conjunction is distributive over disjunction)}
\]
\[
\iff \exists (g_3, S_3) \in [e_1[t]] \wedge \alpha \in g_3 \vee \exists (g_3, S_3) \in [e_2[t]] \wedge \alpha \in g_3 \quad \text{(using Lemma 4.4)}
\]
\[
\iff \exists (g_3, S_3) \in [(e_1[t] \times e_2[t])[t]] \wedge \alpha \in g_3 \quad \text{(using definition of CROSS)}
\]
\[
\iff \exists (g_4, S_4) \in [(e_1[t] \times e_2[t])[t]] \wedge \alpha \in g_4 \quad \text{(using Lemma 4.4 and the fact that} \alpha \leq t).
\]

Triple Pattern Push-down theorem allows performing triple pattern matching before computing the cross product of two expressions. This can reduce the size of the two sets to be crossed and therefore reduces the cost of evaluation. This is particularly important in distributed environments as computing Cartesian product between two sets is typically a costly operation. It is worth pointing out that this property is similar to filter push-down in relational algebra.

Notice that in Theorem 4.9, applying the triple pattern $t$ on the result of the cross product of $e_1[t]$ and $e_2[t]$ is essential for equivalence. In other words, $(e_1 \times e_2)[t] \neq (e_1[t] \times e_2[t])$, because the two expressions will have different binding components.

The next theorem generalises triple pattern push-down to allow pushing arbitrary patterns as long as they are less specific than the pattern in the original expression.

**Theorem 4.10.** If $e_1, e_2$ are two RDF.co expressions and $t, t_1, t_2$ are three triple patterns such that $t \leq t_1$ and $t \leq t_2$, then 
\[(e_1 \times e_2)[t] \equiv (e_1[t_1] \times e_2[t_2])[t].\]
Proof. We make use of Corollary 4.7 in this proof.

We first prove that if \( \alpha \in g_1 \) for some \( (g_1, S_1) \in \llbracket (e_1 \times e_2)[t] \rrbracket \), then \( \exists (g_2, S_2) \in \llbracket (e_1[t_1] \times e_2[t_2])[t] \rrbracket \) such that \( \alpha \in g_2 \).

\[
\alpha \in g_1 \land (g_1, S_1) \in \llbracket (e_1 \times e_2)[t] \rrbracket
\implies \exists (g_2, S_2) \in \llbracket e_1 \times e_2 \rrbracket \land \alpha \in g_2 \land \alpha \leq t \quad \text{(using Lemma 4.4)}
\implies ((g_2, S_2) \in \llbracket e_1 \rrbracket \land \alpha \in g_2 \land \alpha \leq t) \lor ((g_2, S_2) \in \llbracket e_2 \rrbracket \land \alpha \in g_2 \land \alpha \leq t) \quad \text{(using definition of CROSS)}
\]

As \( \leq \) is transitive, and given that \( t \leq t_1 \) and \( t \leq t_2 \), we can infer that \( \alpha \leq t_1 \) and that \( \alpha \leq t_2 \).

\[
\implies ((g_2, S_2) \in \llbracket e_1 \rrbracket \land \alpha \in g_2 \land \alpha \leq t_1) \lor ((g_2, S_2) \in \llbracket e_2 \rrbracket \land \alpha \in g_2 \land \alpha \leq t_2)
\implies (\exists (g_3, S_3) \in \llbracket e_1[t_1] \rrbracket \land \alpha \in g_3) \lor (\exists (g_3, S_3) \in \llbracket e_2[t_2] \rrbracket \land \alpha \in g_3) \quad \text{(using Lemma 4.4)}
\implies \exists (g_3, S_3) \in \llbracket e_1[t_1] \times e_2[t_2] \rrbracket \land \alpha \in g_3 \quad \text{(using definition of CROSS)}
\implies \exists (g_4, S_4) \in \llbracket (e_1[t_1] \times e_2[t_2])[t] \rrbracket \land \alpha \in g_4 \quad \text{(using Lemma 4.4 and because \( \alpha \leq t \)).}
\]

This proves the first direction of the equivalence as stated in Corollary 4.7. We next prove the other direction.

\[
\alpha \in g_1 \land (g_1, S_1) \in \llbracket (e_1[t_1] \times e_2[t_2])[t] \rrbracket
\implies \exists (g_2, S_2) \in \llbracket e_1[t_1] \times e_2[t_2] \rrbracket \land \alpha \in g_2 \land \alpha \leq t \quad \text{(using Lemma 4.4)}
\implies (\exists (g_2, S_2) \in \llbracket e_1[t_1] \rrbracket \land \alpha \in g_2 \land \alpha \leq t) \lor (\exists (g_2, S_2) \in \llbracket e_2[t_2] \rrbracket \land \alpha \in g_2 \land \alpha \leq t) \quad \text{(using definition of CROSS)}
\implies (\exists (g_3, S_3) \in \llbracket e_1 \rrbracket \land \alpha \in g_3 \land \alpha \leq t_1 \land \alpha \leq t) \lor (\exists (g_3, S_3) \in \llbracket e_2 \rrbracket \land \alpha \in g_3 \land \alpha \leq t_2 \land \alpha \leq t)
\implies \exists (g_3, S_3) \in \llbracket e_1 \times e_2 \rrbracket \land \alpha \in g_3 \land \alpha \leq t \quad \text{(using definition of CROSS and the fact that conjunction is distributive over disjunction)}
\implies \exists (g_4, S_4) \in \llbracket (e_1 \times e_2)[t] \rrbracket \land \alpha \in g_4 \quad \text{(using Lemma 4.4)}.
\]

The generalised form of Triple Pattern Push-down can enable further optimisations. In addition to potential reduction in the size of the crossed sets, it can
enable re-use of previously computed results in a sense similar to that enabled by
Triple Pattern Insertion Theorem (i.e., Theorem 4.8).

4.2 Evaluating Graph and Binding Components
of an Expression

Values of \texttt{RDF.co} expressions have two components: graph and binding compo-

nents. As discussed before, having the two components is necessary to allow
arbitrary cascading of various \texttt{RDF.co} expressions. Nevertheless, having the two
components introduces an extra cost for computation and storage. In this sec-

tion, we discuss mitigating this extra cost. We first introduce component-based
equivalence between expressions.

4.2.1 Component-based Equivalence between \texttt{RDF.co} Ex-
pressions

Definition 4.3. Two expressions \( e_1 \) and \( e_2 \) are said to be \( b \)-equivalent when
\( e_1.b \equiv e_2.b \) We denote \( b \)-equivalence by writing \( e_1 \equiv_b e_2 \).

Definition 4.4. Two expressions \( e_1 \) and \( e_2 \) are said to be \( g \)-equivalent when \( e_1.g \equiv e_2.g \) We denote \( g \)-equivalence by writing \( e_1 \equiv_g e_2 \).

Notice that while equivalence between two \texttt{RDF.co} expressions implies both \( b \)-
equivalence and \( g \)-equivalence, the other direction is not necessarily true. For
example, if the values of two expressions \( e_1 \) and \( e_2 \) are:

\[
\begin{align*}
[e_1] &= \{ (g^\phi, S^\phi), (\{(s, p, o)\}, (1)) \} \\
[e_2] &= \{ (g^\phi, (1)), (\{(s, p, o)\}, S^\phi) \}
\end{align*}
\]

Then:

\[
\begin{align*}
[e_1.g] &= \{ (g^\phi, S^\phi), (\{(s, p, o)\}, S^\phi) \} \\
[e_2.g] &= \{ (g^\phi, S^\phi), (\{(s, p, o)\}, S^\phi) \} \\
[e_1.b] &= \{ (g^\phi, S^\phi), (g^\phi, (1)) \} \\
[e_2.b] &= \{ (g^\phi, (1)), (g^\phi, S^\phi) \}
\end{align*}
\]

Notice that \( e_1 \equiv_g e_2 \) and \( e_1 \equiv_b e_2 \) but \( e_1 \neq e_2 \).
4.2.2 Dependencies between Components of RDF.co Expressions Values

We aim to mitigate the extra cost of maintaining two components in each RDF.co expression value by:

- Identifying operators that do not change one of the two components (i.e., the graph or the binding) and, therefore, allow skipping some computation.

- Studying dependency between different components when computing the value of a particular operator. If for example, the binding component of an expression is not used to compute the results of further operators, computing and storing the binding component can be skipped.

Figure 4.2 shows the dependency between the components of a value of an expression and the components of the results of applying an RDF.co operator. A dependency between two components means that changing the input component can result in a change of the value of the result component. We capture the dependency by a dashed arrow in the figure. A special type of dependency between two components is equality. We capture equality by a solid arrow in the figure. Notice that the resulting graph of each operator is always a bipartite graph. Notice also that the dependencies can be extracted by examining the semantic definition of each of the operators (Definition 3.2).

We next introduce a number of definitions based on the notation of Figure 4.2.

**Definition 4.5.** If the b component (g component) in the result has only an exact arrow connecting to it, we say that the operator is b-idempotent (g-idempotent, respectively).

**Definition 4.6.** If all arrows are between components of the same type (i.e., b to b or g to g), then we say that the operator is component-separable.

**Definition 4.7.** If the b component (g component) of the input has no arrow going out of it, we say that the operator is b-independent (g-independent, respectively).

Based on the definitions above:
Figure 4.2: Analysis of the dependency of RDF.co expressions values. $g$ and $b$ are the graph and binding components of the input while $g'$ and $b'$ are the graph and binding components of the result of applying the expression to the input.

- $b\text{Access}$ is $g$-independent, $b$-idempotent, and component-separable.
- $\text{PROJ}$ is $g$-idempotent and component-separable.
- $b\text{Ext}$ is $g$-idempotent and component-separable.
- $g\text{Ext}$ is $b$-idempotent.
- $\text{MATCH}$ is $b$-independent.
- $\text{CROSS}$ is component-separable.
- $\text{FILTER}$ has none of the above characteristics.

To illustrate the consequences of each of the operator characteristics, we introduce a general notation to indicate applying any of the RDF.co operators to an expression. If $\rho$ is an RDF.co operator and $e$ is an RDF.co expression, we refer to the expression that results from applying $\rho$ to $e$ as $\rho(e)$. Consequently:
• If $\rho$ is b-idempotent then $\rho(e) \equiv_b e$. This means that when evaluating $\rho(e)$, the binding component of the result is the same as the binding component of the input when $\rho$ is b-idempotent.

• If $\rho$ is g-idempotent then $\rho(e) \equiv_g e$. This means that when evaluating $\rho(e)$, the graph component of the result is the same as the graph component of the input when $\rho$ is g-idempotent.

• If $\rho$ is b-independent then $\rho(e.g) \equiv \rho(e)$. This means that when evaluating $\rho(e)$, the binding component of the input can be ignored and $\rho$ can be evaluated based only on the graph component of the input when $\rho$ is b-independent.

• If $\rho$ is g-independent then $\rho(e.b) \equiv \rho(e)$. This means that when evaluating $\rho(e)$, the graph component of the input can be ignored and $\rho$ can be evaluated based only on the binding component of the input when $\rho$ is g-independent.

• If $\rho$ is component-separable then $\rho(e) \equiv_b \rho(e.b) \wedge \rho(e) \equiv_g \rho(e.g)$. This means that, when $\rho$ is component-separable, the binding component of the result depends only on the binding component of the input and, similarly, the graph component of the result depends only on the graph component of the input. This can be utilised when parallelising the computation of the two components as no data transfer is needed between the two computations.

When applying the results to the RDF.co operators we get:

• bAccess:
  - $e \equiv_b e.b$ ($\text{bAccess}$ is b-idempotent).
  - $(e.b).b \equiv e.b$ ($\text{bAccess}$ is g-independent).
  - $(e.b) \equiv_b (e.b).b$ and $(e.b) \equiv_g (e.g).b$ ($\text{bAccess}$ is component-separable).

• PROJ:
  - $e|_{(a_1,\ldots,a_n)} \equiv_g e$ ($\text{PROJ}$ is g-idempotent).
  - $e|_{(a_1,\ldots,a_n)} \equiv_b (e.b)|_{(a_1,\ldots,a_n)}$ and $e|_{(a_1,\ldots,a_n)} \equiv_g (e.g)|_{(a_1,\ldots,a_n)}$ ($\text{PROJ}$ is component-separable).

• bExt:
Chapter 4. RDF.co

- \( e \oplus_{(a_1,\ldots,a_n)} h \equiv_{g} e \) (\( b\text{EXT} \) is \( g \)-idempotent).
- \( e \oplus_{(a_1,\ldots,a_n)} h \equiv_{b} (e.b) \oplus_{(a_1,\ldots,a_n)} h \) and \( e \oplus_{(a_1,\ldots,a_n)} h \equiv_{g} (e.g) \oplus_{(a_1,\ldots,a_n)} h \) (\( b\text{EXT} \) is component-separable).

- \textbf{gExt}: \( (e \oplus (\alpha_1,\alpha_2,\alpha_3)) \equiv_{b} e \) (\( g\text{EXT} \) is \( b \)-idempotent).

- \textbf{MATCH}: \( (e.g)[t] \equiv e[t] \) (\( MATCH \) is \( b \)-independent).

- \textbf{CROSS}: \( (e_1 \times e_2) \equiv_{b} (e_1.b \times e_2.b) \) and \( (e_1 \times e_2) \equiv_{g} (e_1.g \times e_2.g) \) (\( CROSS \) is component-separable).

The previous properties can be utilised to simplify RDF.co expressions. We next list few example expressions and possible rewritings.

- \( e = (e_1 \times e_2)[(s,p,\,?)]) \)
  
  \( e \equiv ((e_1 \times e_2).g)[(s,p,\,?)] \) (\( MATCH \) is \( b \)-independent)

  As \( CROSS \) is component-separable, we know that \( (e_1 \times e_2) \equiv_{g} (e_1.g \times e_2.g) \) which can be rewritten using the definition of \( g \)-equivalence as: \( (e_1 \times e_2).g \equiv (e_1.g \times e_2.g).g \)

  Therefore, \( e \equiv ((e_1.g \times e_2.g).g)[(s,p,\,?)] \)

  The previous rewriting of \( e \) eliminates the need to compute the result of crossing the binding components of \( e_1 \) and \( e_2 \). In other words, it is sufficient to cross only the graph components to produce the final results. This can significantly reduces the cost of evaluating the expression \( e \).

- \( e = (e_1[(s,p,\,?)] \oplus (s_1,p_1,1)).b \)
  
  \( e \equiv (e_1[(s,p,\,?)]).b \) (\( g\text{EXT} \) is \( b \)-idempotent).

  The previous rewriting of \( e \) eliminates the need to compute the \( g\text{EXT} \) operator.

- \( e = (e_1[(s,p,\,?)] \oplus_1 h).g \)
  
  \( e \equiv (e_1[(s,p,\,?)]).g \) (\( b\text{EXT} \) is \( g \)-idempotent).

  The previous rewriting of \( e \) eliminates the need to compute the \( b\text{EXT} \) operator.
• \( e = (e_1[(s,p,?)] \oplus_1 h).b \)

\[ e \equiv ((e_1[(s,p,?)]) . b \oplus_1 h).b \quad (\text{\texttt{bEXT} is component-separable}) \]

The previous rewriting of \( e \) eliminates the need to compute the graph components of the \texttt{MATCH} operator.

4.3 Summary

In this chapter we studied a number of unique algebraic properties of the \texttt{RDF.co} expressions. We focused on the two aspects of \texttt{RDF.co} that set it different from the well-studied SPARQL algebra; namely, cascading triple patterns and pairing graphs and bindings in each value. These algebraic properties can be utilised in optimising query plans when evaluating \texttt{RDF.co} expressions. In the next chapter, we describe extending \texttt{RDF.co} with aggregate operators.
Chapter 5

Extending RDF.co with Aggregation Operators

Aggregating data into concise summaries is an essential part of On-Line Analytical Processing [38] (OLAP) and decision support applications [31]. Therefore, relational algebra provides rich support for data aggregation. In addition to the SQL aggregate functions and the GROUP BY operator, many relational databases support the cube operator [64], graph aggregation [33, 177], and more expressive aggregations [29, 30]. Given the importance of data aggregation, the latest version of SPARQL, SPARQL 1.1, added support for aggregates.

In this chapter we define two aggregation operators as part of RDF.co. The first operator provides aggregation capabilities similar to that of relational algebra and SPARQL (Section 5.1), whereas the second operator allows condensing graphs into smaller aggregated graphs (Section 5.2). In Section 5.3, we discuss the effects of having the aggregation operators on properties of RDF.co expressions presented in the previous two chapters.

5.1 Aggregation Operators (AGG)

Syntax:

If $e$ is an RDF.co expression, $d$ and $m$ are two natural numbers and $f$ is a function symbol then $(e \langle d, f, m \rangle)$ is an RDF.co expression.
Semantics:

The expression \((e \langle d, f, m \rangle)\) groups by the \(d^{th}\) element in the binding, then apply the aggregate function \(f\) on the values of the \(m^{th}\) element in each group. In other words, the \(d^{th}\) element in the binding defines the dimension of aggregation while the \(m^{th}\) element defines the measure to be aggregated. Therefore, \(f\) is interpreted as a function that takes a multi-set of terms and returns a single value (i.e., the signature of an aggregate function is \(f : \mathcal{P}(\mathcal{T}) \rightarrow \mathcal{T}\)).\(^1\) We use the same set of aggregate functions that are defined by SPARQL (Section 18.5 in [72]). Furthermore, in the presence of errors, we assume that these functions behave in the same way defined in the SPARQL specifications.

We first define an auxiliary function \(\text{bfilter}\) that given a set \(R\) of pairs of graph and binding, a value \(v\), and two indices \(d\) and \(m\); extracts from the set \(R\) the bindings that have the value \(v\) in their \(d^{th}\) position and then returns a multiset of the values in the \(m^{th}\) position of the filtered bindings.

\[
\text{bfilter} : \mathbb{N} \times \mathcal{T} \times 2^{G \times S} \times \mathbb{N} \rightarrow \mathcal{P}(\mathcal{T}) : \text{bfilter}(d, v, R, m) = \bigcup_{\{(g, S)\in R, S_d = v\}} \{\{S_m\}\}^2
\]

Using the function \(\text{bfilter}\), we define the semantics of the \textbf{AGG} expressions:

\[
[(e \langle d, f, m \rangle)] = \bigcup_{(S_d \ni (g, S) \in [e])} (g^\phi, (S_d, f(\text{bfilter}(d, S_d, [e], m))))
\]

See Figure 5.1 for an example and notice that the resulting RDF graphs are empty (absence of graph in the figure indicates empty graph component). Notice that in the example of Figure 5.1, \(\text{bfilter}\) will be applied twice as follows:

\[
\begin{align*}
\text{bfilter}(1, o, [e], 2) &= \{\{10, 3\}\} \\
\text{bfilter}(1, v, [e], 2) &= \{\{2\}\}
\end{align*}
\]

\(^1\) For simplicity of the presentation here, we restrict aggregate functions to those that take a set of single values. Generalising this for aggregate functions with more than one parameter is straightforward. We use \(\mathcal{P}(\cdot)\) to indicate the powerset with duplicates.

\(^2\) We use \(\{\{}\\}\) to indicate a multiset.
5.2 Graph Aggregation Operator (Gagg)

Graph aggregation condenses a large graph into a structurally similar but smaller graph by collapsing vertices and edges. Aggregating RDF graphs has been used to induce schemas [22, 93, 104], produce descriptive statistics [13, 21, 113] and build indices [92, 120].

While SPARQL has support for aggregation, it does not have an operator for graph aggregation. A number of other systems [33, 159, 176, 177] support graph aggregation. We extend RDF.co by Gagg, an operator for graph aggregation.

Gagg is defined in two-steps as shown in Figure 5.2. Firstly, nodes and edges in the original graph $G$ are grouped together in an intermediate graph that we call the grouped graph. Secondly, the grouped graph is reduced into an aggregated graph.

The first step is based on a set of dimensions that defines how nodes and edges are grouped together. The second step condenses the grouped graph into the final aggregated graph. The graph condensation is done by simultaneously applying three aggregation functions on subjects, properties, and objects. It is worth pointing out that the grouped graph is not an RDF graph as its nodes represent sets of resources, while the aggregated graph is an RDF graph that can be used as input for further processing. Corresponding bindings are populated based on aggregation keys and aggregated values.

Syntax:
If $e$ is an RDF.co expression, $A_d, A_m, B_d, B_m, C_d, C_m$ are natural numbers, and $f_A, f_B, f_C$ are function symbols then $(e(A_d, f_A, A_m, B_d, f_B, B_m, C_d, f_C, C_m))$ is an RDF.co expression.

**Semantics:**

We first extend the function `bfilter` defined in the previous section to be able to filter based on multiple values instead of one. Recall that the function `bfilter` when given a set $R$ of pairs of graph and binding, a value $v$, and two indices $d$ and $m$; extracts from the set $R$ the bindings that have the value $v$ in their $d$th position and then returns a multiset of the values in the $m$th position of the filtered bindings. We overload the function `bfilter` to filter based on multiple values as follows:

$$
\text{bfilter}(a_1, v_1, a_2, v_2, a_3, v_3, R, m) = \bigcup_{(g, S) \in R, S_{a_1} = v_1 \land S_{a_2} = v_2 \land S_{a_3} = v_3} \{\{S_m\}\}
$$

Next we define the semantics of $Gagg$:

$$
[(e(A_d, f_A, A_m, B_d, f_B, B_m, C_d, f_C, C_m))] =
$$

$$
\bigcup_{(S_{A_d}, S_{B_d}, S_{C_d}) \exists (g, S) \in [e]} \{(X, S_{B_d}, Y)\},
$$

$$
S_{A_d}, f_A(bfilter(A_d, S_{A_d}, B_d, S_{B_d}, C_d, S_{C_d}, [e], A_m)),
$$

$$
S_{B_d}, f_B(bfilter(A_d, S_{A_d}, B_d, S_{B_d}, C_d, S_{C_d}, e[], B_m)),
$$

$$
S_{C_d}, f_C(bfilter(A_d, S_{A_d}, B_d, S_{B_d}, C_d, S_{C_d}, [e], C_m))
$$

Where $X$ and $Y$ in the previous definition are two fresh unique IRIs.
Figure 5.3 shows an example evaluation of a Gagg expression and Figure 5.4 diagrammatically explains the multiple applications of \texttt{bfilter} function during the evaluation. In Figure 5.3, \texttt{Gagg} is grouping resources that are in column 1 and those that in column 3 and counting the number of predicates in column 2 between each two groups.

![Figure 5.3: Gagg example](image)

### 5.3 Properties of the Aggregation Operators

#### 5.3.1 Length of Binding

Lemma 3.2 states that all bindings in an RDF.co expression have the same length. Consequently, a notion of length of binding of expressions was introduced. Lemma 3.2 still holds with the addition of aggregation operators to RDF.co set of operators. In particular, all bindings in the value of an AGG expression have a length equal to the number of dimensions and measures in the expression. The bindings in the value of AGG expressions have an entry for each of the dimensions and one for each measure. Similarly, all bindings in the value of an Gagg expression have a length equal to the total number of dimensions and measures in the expression.
5.3.2 Primitivity of the Aggregation Operators

AGG expressions can be expressed using a combination of Gagg, PROJ, and bAccess. The idea is to use the same dimension and measure of the AGG operator three times in a Gagg expression. Afterwards, PROJ and bAccess operators are necessary to format the binding and the graph values to match AGG results.

\[ e \{d, f, m\} \equiv ((e \{d, f, m, d, f, m, d, f, m\})_{(1,2)}.b \]

The equivalence results from the fact that the extension of the bfilter function for Gagg expressions reduces to the original one when \(a_1 = a_2 = a_3\) and \(v_1 = v_2 = v_3\).

On the other hand, Gagg is not primitive as it has the unique ability to create fresh values per result. Furthermore, AGG expressions cannot be expressed without Gagg (i.e., using the RDF.co operators of Chapter 3) as all other operators cannot group multiple items in the results together.
5.3.3 Expressing SPARQL Aggregation

The addition of \texttt{AGG} to the RDF\texttt{.co} extends its ability to express aggregation expressions of SPARQL 1.1. A recent formal study of SPARQL aggregation was presented in [87]. In the SPARQL algebra, aggregate queries are defined via the operators Group, Aggregate, and AggregateJoin [72, 87]. However, [87] showed that a simpler normalised form of aggregate queries can be defined. The normalised form is similar to that of relational algebra and to the one we defined for RDF\texttt{.co}. The difference between SPARQL aggregate queries as defined in [87] and our definition is that for SPARQL a pattern is required as part of the aggregate query definition. In RDF\texttt{.co}, any expression can be defined before applying aggregation. This difference stems from the original motivation to allow arbitrary cascading in RDF\texttt{.co} while the SPARQL algebra always requires triple patterns as the basis of any query.

SPARQL, on the other hand, does not provide direct support for graph aggregation. We discuss extending SPARQL with such operator in Chapter 10.

5.3.4 Cascading RDF\texttt{.co} Operators

Similar to all other RDF\texttt{.co} operators, aggregation operators can be arbitrarily cascaded with other operators. This results from the consistent form of the input and output of all RDF\texttt{.co} expressions, including the aggregation ones.

5.3.5 Dependencies Between Graph and Binding Components

Figure 5.5 shows the dependencies between graph and binding components when applying AGG and Gagg operators. Using the definitions in Section 4.2.2, we can infer that AGG is component-separable and g-independent whereas Gagg is g-independent. Using the formal notation:

- \( e \langle d, f, m \rangle \equiv (e.b) \langle d, f, m \rangle \) (AGG is g-independent.)
- \( e \langle d, f, m \rangle \equiv_b (e.b) \langle d, f, m \rangle \) and \( e \langle d, f, m \rangle \equiv_g (e.g) \langle d, f, m \rangle \) (AGG is component-separable.)
Chapter 5. RDF.co

5.4 Summary

In this chapter we extended RDF.co with two aggregation operators: AGG, a relational-like aggregation operator, and Gagg, a graph aggregation operator. This adds an important capability to RDF.co. Aggregation is particularly necessary in analytics and business intelligence contexts, a common need when processing large amounts of data. We also revisited properties of RDF.co expressions discussed in previous chapters in the light of the addition of the new operators. This chapter concludes the definition of RDF.co, our data model to support a dataflow language for processing large RDF graphs.
Part II

Graph Partitioning, the Physical Data Layout
Chapter 6

Pattern-based Graph Partitioning

As discussed in the Introduction chapter, distributed platforms are being utilised to process large graph data. The MapReduce paradigm was used to process graphs with hundreds of millions of nodes and edges (e.g., [48, 81, 88]). Furthermore, a number of specialised distributed graph systems, based on vertex-centric [63, 114, 140, 172] and more recently on graph-centric [137, 147, 158, 173] computation model, are also used.

In all these shared-nothing systems, the performance bottleneck is the communication over the network. Therefore, effective partitioning and placement of graph data is of paramount importance [63, 105, 116]. Current practices to graph data partitioning range from random placement of nodes to finding a min-cut partitioning (i.e., a partitioning that minimises the number of edges that cross machines).

Recently, graphs are increasingly used as an integration vehicle where data from multiple sources and domains are put together\(^1\). This commonly results in heterogeneous large graphs as can be seen in graphs produced by the Semantic Web community, such as WikiData\(^2\) and DBpedia\(^3\), and graphs used by Internet companies such as the Google Knowledge Graph\(^4\) and the Facebook Social Graph. In this chapter, we argue that this increasing heterogeneity calls for a rethinking of graph partitioning that departs from the traditional min-cut algorithms.

---

\(^1\)These graphs are also referred to as Heterogeneous Information Networks in literature.
\(^2\)http://www.wikidata.org
\(^3\)http://dbpedia.org
\(^4\)http://googleblog.blogspot.fr/2012/05/introducing-knowledge-graph-things-not.html
Chapter 6. Pattern-based Graph Partitioning

As an integration medium, graph data comes from different sources, various domains, and describes multiple types of entities and relationships. Hence, different edges in the graph have different importance in different scenarios. Furthermore, related entities in a graph are not necessarily connected by direct edges. Consequently, we argue for, and propose, a more flexible formulation of the graph partitioning problem. In particular, we argue for using pattern matching to guide graph partitioning. Using pattern matching, knowledge that might be available about the data or the task at hand can be leveraged to guide the partitioning strategy.

After reviewing graph partitioning methods in existing distributed systems in Section 6.1, we discuss the use of pattern matching as a vehicle to guide graph partitioning in Section 6.2. Section 6.3 discusses related work and shows that pattern-based formulation generalises common partitioning methods reported in previous literature, facilitating a unified way to compare and study them.

6.1 Graph Partitioning and Distributed Platforms

In shared-nothing distributed systems, a program comprises a number of tasks running in parallel. Each of these tasks handles a subset of the data. The portion of the data that a task processes varies among platforms. It is a \( (key, value) \) pair in MapReduce; a single vertex and its neighbours in vertex-centric platforms such as Pregel [114] and GPS [140]; a subgraph in Giraph++ [158] and NScale [137]; a connected subgraph in Blogel [173] and Goffish [147]; and a star-shaped graph with all the nodes up to \( n \) hops away from the centre in systems with \( n \)-hops guarantees [81, 99].

Interleaving sequential and parallel computation is one important innovation of recent distributed computation platforms. In MapReduce [46], a program is divided into a series of two sequential steps, a map and a reduce. Data communication between tasks occurs only when map tasks are finished and before reduce tasks start. Similarly, in Bulk Synchronous Parallel (BSP) systems that are inspired by Pregel [114], the program is divided into a number of sequential supersteps. Each of these supersteps runs a number of vertex-centric tasks in parallel. Communication takes place only after each superstep in order to ensure scalability and fault tolerance.
Hence, maximising the work each task can do before it needs to communicate with other tasks is a key for efficient distributed programs. It is, therefore, instrumental to partition the data in a way that increases data locality provided for each task. For instance, in Blogel [173], finding strongly connected components was run first by each task locally on its portion of the data before the results get combined. The fact that, in Blogel, the portion of data assigned to each task is a connected subgraph guarantees the correctness of the algorithm. This dramatically reduced the size of exchanged messages and the number of needed iterations. Similarly, in [81], many SPARQL queries were answered intra-partition (i.e., with no network communication) when the radius of the query is smaller than the number of hops that was used in partitioning. Generally, matching the partitioning strategy to the expected workload in distributed processing systems can have tremendous effect on performance. This is similar to the conventional wisdom of relational databases that encourages matching index structure to the expected workload.

Therefore, when answering pattern-based queries or computing some graph measures based on patterns, we argue that it is beneficial to guide the data partitioning using patterns. In the next section, we describe pattern-based graph partitioning and discuss related challenges.

### 6.2 Pattern Matching for Partitioning RDF Graphs

Pattern matching is one important way of querying heterogeneous graphs. Pattern matching provides a declarative description of the parts of graphs that are of interest. It can be used to find neighbours of a node, fixed-length paths, and to extract relevant portions of a graph. Pattern matching is the core of graph query languages such as SPARQL [72], the W3C recommended language for querying RDF data, and Cypher\(^5\), Neo4j query language. In RDF.co we proposed in Chapter 3, pattern matching was also an essential part in defining the operators MATCH and GMATCH.

We use the example graph shown in Figure 6.1 to illustrate pattern-based partitioning. We represent patterns with a graph where nodes and edges can be variables (indicated by a preceding ’?’). Notice that this chapter provides only

an illustrative description, whereas formal definitions and related algorithms are described in the next chapter.

Intuitively, the goal of partitioning a graph according to a given pattern is to ensure that each set of nodes and edges of the graph that matches the pattern are all included together in some partition. Therefore, to partition an RDF graph using a given pattern, we match the pattern against the graph and then group the results into partitions based on some designated node in the pattern. In a partition, the node used for grouping the triples is referred to as the root node. The following examples demonstrate this.

**Star-shaped partitioning** This partitioning groups triples by their subjects (i.e., each partition constitutes of triples that share the subject and, therefore, is shaped as a star graph). This can be achieved by matching the graph against the pattern:

\[ ?X \rightarrow ?E \rightarrow ?Y \]

Notice that the node to group triples by is underlined (?X in the previous pattern). Corresponding partitions of the example graph are shown in Figure 6.2.

**2-hops partitioning** Similar to star-shaped partitioning but also includes nodes that are two hops away (i.e., include neighbours and neighbours of neighbours). This can be achieved by matching the graph against the pattern:

\[ ?X \rightarrow ?E \rightarrow ?Y \rightarrow ?F \rightarrow ?Z \]
Figure 6.2: Example partitioning of the graph in Figure 6.1 into star subgraphs

Corresponding partitions of the example graph are shown in Figure 6.3. Notice that the obtained subgraphs are overlapping. In our example data, the size of the subgraph rooted at the resource labelled John is almost equal to the size of the total graph. Moreover, notice that the semantic of matching we use when partitioning the data includes neighbours of neighbours when they exist but do not exclude neighbours that do not have further neighbours. This is essential to ensure that each triple is assigned to at least some partition.

Equivalent resources Edges labelled with sameAs in Figure 6.1 indicate equivalence between nodes. In some scenarios, it can be beneficial to place equivalent nodes in the same partition even when min-cut or n-hop partitioning suggests otherwise. This can be achieved using the following pattern:
(A) The 2-hops subgraph rooted at John

(B) The 2-hops subgraph rooted at Ronan

(C) The 2-hops subgraph rooted at Sarah

(D) The 2-hops subgraph rooted at Peter

(E) The 2-hops subgraph rooted at Johny

(F) The 2-hops subgraph rooted at Paddy

**Figure 6.3:** Example partitioning of the graph in Figure 6.1 into 2-hops subgraphs
Common friends To ensure that people who share a common friend are always included in the same partition, the following pattern can be used:

Results of applying this pattern to partition the example graph are shown in Figure 6.4. In the figure, it can be seen, for instance, that the triple (John, friend, Sarah) is included in the subgraph rooted at Sarah because John and Sarah share a friend. Furthermore, it is worth reiterating here that subgraphs are overlapping.

6.3 Related Work

In this section we compare pattern-based partitioning to two commonly used methods for partitioning graphs, namely min-cut and n-hops partitioning. We then show that pattern-based partitioning can generalise a number of common partitioning methods reported in previous literature and facilitate a unified way to compare and study them.

6.3.1 Min-cut Partitioning

Min-cut partitioning is a commonly used method to partition graphs. In distributed systems, a balanced p-way min-cut achieves two goals:

- Partitions are of similar sizes in order to achieve load balancing.
- The number of crossing edges (i.e., edges that have their ends in different partitions) is minimal in order to reduce communication and storage cost.
Formally a balanced p-way edge-cut can be defined as follows [63]:

$$\min_{A} \frac{|\text{Edge Cuts}|}{|E|} \quad \text{s.t.} \quad \max_{m\in V} \{|v : A(v) = m\} < \frac{|E|}{\lambda p}$$

where $p$ is the number of partitions and $\lambda$ is a small factor $\geq 1$ that represents an imbalance factor.

PowerGraph [63] uses min-cut to minimise the number of vertices cut (i.e., vertices that are assigned to more than one machine) instead of minimising the number of edges cut.

Min-cut partitioning is computationally expensive [96, 100]. It is also oblivious to edge labels and to expected query loads. Therefore, it is challenging to apply min-cut partitioning to heterogeneous large graphs as edges are labelled and have
varying importance in different scenarios. In fact, when the graph is difficult to partition, both GraphLab [108] and Pregel [114] resort to hashed vertex placement (i.e., random vertex placement).

To further motivate pattern-based partitioning, we conducted a small experiment where we partitioned an RDF graph using METIS, a state-of-the-art min-cut graph partitioning system. The RDF graph we used is generated using the LUBM benchmark and contains about 600K triple. We split the data into 5 partitions and measured the number of triples communicated between the partitions during query answering. As shown in table 6.1, min-cut partitioning reduces the communication cost significantly over random partitioning. Nevertheless, the fact that graph partitioning is oblivious to the edge labels and to expected query load makes it suboptimal in some scenarios. The last column in Table 6.1 shows the perfect scenario, by using the exact same query for partitioning. However, in less ideal and more realistic scenarios, when a particular pattern is known to be frequent in the anticipated query load, that pattern can be used to guide partitioning and, therefore, to reduce communication across partitions. For example, the last two rows in Table 6.1 show patterns that occur frequently in the LUBM benchmark.

<table>
<thead>
<tr>
<th>Query</th>
<th>random</th>
<th>METIS</th>
<th>same pattern as the query</th>
</tr>
</thead>
<tbody>
<tr>
<td>?s ?p ?o</td>
<td>17400</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>?s ?p ?o . ?o ?p2 ?s</td>
<td>30500</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>?s ?p ?o . ?o ub:subOrganization ?o2</td>
<td>4300</td>
<td>1700</td>
<td>0</td>
</tr>
<tr>
<td>?s ?p ?o . ?o rdf:type ?o2</td>
<td>26000</td>
<td>8800</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1: Number of triples sent over the network while query answering using different partitioning approaches

6.3.2 n-hops Partitioning

Nodes that are neighbours to a particular node are said to be 1-hop away from that node. Similarly, neighbours of neighbours are 2-hop away and so on. The state-of-the-art of RDF processing on top of MapReduce exploits this graph structure to partition the RDF data [81, 99].
For example in 1-hop partitioning, each graph partition is centred at a particular node and includes all its direct neighbours. Notice that all star queries can be answered intra-partition in this case, i.e., without the need of any communication across partitions. In order to be able to answer more complicated queries, graph-based partitioning expands the partitions to include further neighbours. These methods provide what is called \textit{n-hops guarantees} that include all nodes up to \( n \) hops away from each star-graph centre \([81, 99]\).

It is worth pointing out that in comparison to the n-hop guarantees, pattern matching allows better control on the level of replication. The n-hop guarantee is an exhaustive method that replicates all edges within some distance from a node. Patterns, on the contrary, allow selective replication and support, therefore, more control over the replication-communication trade-off.

### 6.3.3 Defining Existing Partitioning Methods Using Graph Patterns

In this section, we show that pattern-based partitioning can represent a number of common partitioning methods reported in previous literature. In this regards, pattern-based partitioning provides a unified way to compare and study these partitioning methods. Notice that we distinguish the variables that are used to define the root by underlying them in the figures.

**Vertical partitioning** \([4]\): This method partitions edges based on their labels.

\[ X \xrightarrow{E} Y \]

This partitioning method corresponds to a query load that uses short paths with bound predicates.

**Ego-network partitioning:** This is also called star-shaped graph partitioning as each partition constitutes of a set of edges that share the source and, therefore, is shaped as a star graph. This method is referred to in \([81]\) and \([99]\) as directed 1-hop guarantee. It is also one of the exhaustive indices used by single-node triple stores such as Hexastore \([169]\) and RDF-3X \([124]\). This partitioning method suits a query load of star-queries that consist of multiple patterns that all share one node.
Directed 2-hop guarantee partitioning [81, 99]: Similar to ego-network partitioning but also includes nodes that are two hops away.

Undirected 1-hop guarantee partitioning [81, 99]: Similar to ego-network partitioning but includes both inbound and outbound edges.

Other partitioning methods such as 2D hash partitioning [23] relies on hashing more than one resource. 2D hash partitioning determines the partition an edge belongs to by hashing its two ends. While such partitioning can be described using patterns as shown below, our formal model and algorithms (described in the next chapter) covers only the case where partitioning are based on extending star-shaped graphs.

6.4 Summary

In this chapter we proposed using pattern matching for graph partitioning. We compared pattern-based partitioning to other graph partitioning algorithms and showcased the generality of pattern matching by using it to express common partitioning methods reported in previous literature. Furthermore, this chapter presented an informal description of pattern-based partitioning. In the next chapter, we formally describe pattern-based partitioning and describe algorithms for implementing data partitioning and query answering using pattern matching.
Chapter 7

Partitioning RDF Graphs

The previous chapter motivated the use of pattern matching for graph partitioning. In this chapter, we discuss the challenges, and present pertinent solutions, when partitioning RDF data for distributed platforms. The example social graph shown in Figure 7.1 is used to motivate the problem and to highlight the involved challenges. When studying common friends, one might partition the data so that people who share friends are included in the same partition. The pattern shown in Figure 7.2a can be used to capture people with common friends. However, supporting the use of such a pattern for partitioning RDF data requires addressing two main tasks:

**Data partitioning and placement:** How is the graph partitioned into subgraphs according to the used pattern? How are these subgraphs placed together in partitions in order to ensure load balancing (i.e., partitions are of similar sizes) and to minimise duplication across partitions? Section 7.2.1 provides a formal definition of pattern-based partitioning. A corresponding algorithm for data partitioning is presented in Section 7.2.2. For data placement, Section 7.2.3 outlines using clustering in order to reduce duplication across different partitions.

**Query answering:** This involves decomposing a given query into a set of subqueries in order to benefit from data locality and to minimise communication across partitions. For instance, Figure 7.2 shows a partitioning pattern and two similar queries. However, despite their similarity, only one of these two queries can be answered with no communication between graph partitions.
resulting from the pattern shown in Figure 7.2a. Section 7.3 shows how deciding on query answerability can be mapped to the well-studied problem of view-based query answering.

We discuss related work in Section 7.4. A MapReduce-based implementation and an experimental evaluation are described in the next chapter.

### 7.1 Preliminaries

#### 7.1.1 Graph Patterns Matching

We use the same notation defined in Section 3.1.1 to define RDF graphs. However, we use named variables for query definition instead of using the symbol "?" and relying on the position of the answer as in `RDF.co`. Both notations are equivalent.

---

1We reuse some notation from [129] in this section.
as one can simply replace variable names by their lexicographical order. Named variables make joined patterns clearer and are therefore used in this chapter.

Therefore, we assume the existence of an infinite set $V$ of variables disjoint from the above sets, and assume that every element in $V$ starts with the symbol ‘?’.

A triple pattern is a tuple from $(T \cup V) \times (I \cup V) \times (T \cup V)$. A graph pattern is a set of triple patterns. The set of variables in a graph pattern $P$ is referred to as $vars(P)$. We refer to [129] for details and formal description of graph pattern evaluation.

Both RDF graphs and graph patterns can be viewed as graphs where each triple represents a directed edge in the graph. Figure 7.1 and Figure 7.2 show examples of an RDF graph and graph patterns, respectively. In the rest of this chapter, we will consider only graph patterns that are connected graphs.

### 7.2 Data Partitioning and Placement

This section details our contribution to define RDF data partitioning and placement based on graph patterns.

#### 7.2.1 Partitioning Scheme

The goal of pattern-based data partitioning is to decompose the total graph into a set of subgraphs according to a given pattern $P$ and to ensure that each of these matching subgraphs is assigned as a whole to a partition. The goal is to ensure that there exists no edge-cuts in the matching subgraphs. The partitioning pattern $P$ is chosen to match the anticipated query load and, consequently, minimising edge-cuts for $P$ enhances query answering time.

Subgraphs matching a given query $P$ can overlap. Hence, a node in the partitioning pattern $P$ is distinguished as the root node and will be denoted as $P_{root}$. $P_{root}$ is used to determine the partition to which matching triples should be assigned. We

\footnote{Queries with disconnected patterns in SPARQL require answering each pattern separately and then performing a full Cartesian product between the results. Hence, each of the disconnected component needs to be evaluated separately and there is, therefore, no loss in generality by considering only connected queries.}
refer to the pair \((P, P_{\text{root}})\) as a partitioning scheme. The next definition captures formally the notion of a partitioning scheme.

**Definition 7.1.** A partitioning scheme is a pair \((P, P_{\text{root}})\) where \(P\) is a graph pattern and \(P_{\text{root}}\) is a variable used in \(P\) (i.e., \(P_{\text{root}} \in \text{vars}(P)\)).

### 7.2.2 Pattern-based Graph Partitioning

An equivalent interpretation of a graph pattern matching can be achieved by decomposing the graph into a set of paths. Matching each of the paths and then joining the results accordingly yield the results of the original query\(^3\). For example, the pattern in Figure 7.2a is equivalent to: \((?x \xrightarrow{r} ?v) \bowtie ?x (\xrightarrow{\text{friend}} ?y \xrightarrow{r} ?z) \bowtie (?y \xrightarrow{\text{friend}} ?z)\)\(^4\).

As stated in the previous chapter, for partitioning, we need to interpret patterns in a more relaxed manner to optionally include further matching triples without removing parts that do not match fully. Decomposing a pattern into a set of paths is therefore useful as it allows matching the paths incrementally. We provide a formal definition of paths and then describe an algorithm that uses the path notion to realise a graph partitioning compatible to a given partitioning scheme.

**Definition 7.2.** A path \(\delta\) rooted at \(o_0 \in (T \cup V)\) is a sequence \(o_0, e_1, o_1, e_2, \ldots, e_n, o_n\) where (i) for \(1 \leq i \leq n\): \(o_i \in (T \cup V)\) and \(e_i\) is either an element in \((I \cup V)\) or \(e_i\) is of the form \(\hat{p}_i\) with \(p_i \in (I \cup V)\) and (ii) no triple \(o_i, e_i, o_{i+1}\) is repeated in \(\delta\)\(^5\).

**Definition 7.3.** A path \(\delta = o_0, e_1, o_1, \ldots, o_n\) contains a triple \((s, p, o)\) if for some \(0 \leq i \leq n - 1\) either \((s, p, o) = (o_i, e_i, o_{i+1})\) or \(e_i\) is of the form \(\hat{p}_i\) and \((s, p, o) = (o_{i+1}, p_i, o_i)\).

**Definition 7.4.** A path \(\delta = o_0, e_1, o_1, e_2, \ldots, e_n, o_n\) belongs to a graph \(G\) (we denote this as \(\delta \in G\)) if and only if all triples contained in \(\delta\) are triples in \(G\).

A path that belongs to a graph represents a sequence of edges in that graph. The notation \(\hat{p}\), borrowed from SPARQL 1.1 property path syntax, captures inverse traversal of a triple with predicate \(p\) (i.e., going from object to subject).

---

\(^3\)This is in fact how the semantic of SPARQL is defined in [129].

\(^4\)We use \(x \xrightarrow{\text{default}} z\) to denote the triple \((x, y, z)\) and \(\bowtie x\) to denote a join based on the variable named \(?x\).

\(^5\)In graph theory terminology, a path without repeated edges is referred to as a trail as well.
Chapter 7. Pattern-based Graph Partitioning

For example, the path Sarah, friend, Ronan, friend, Peter belongs to the RDF graph shown in Figure 7.1 and ?x, friend, ?z, friend, ?y belongs to the graph pattern in Figure 7.2a.

**Definition 7.5.** The set of all paths that are contained in a graph $G$ and rooted at $x$ is denoted as $\text{paths}(G, x)$ (i.e., $\text{paths}(G, x) = \{ \delta | \delta = x, e_1, o_1, \ldots, e_n, o_n \in G \}$).

Notice that the set $\text{paths}(G, x)$ is a finite one as we do not allow repeated edges in a path. Algorithm 1 shows how decomposing a partitioning pattern $P$ into paths can be used to implement a pattern-based partitioning. The algorithm evaluates a path by evaluating each triple pattern in the path against the graph and then joining the results of individual triple patterns. The data structure $T_i$ in the algorithm is used to record the partition and its distinguished node (i.e., the one matching $P_{\text{root}}$). Table 7.1 shows an example join and the use of the $T_i$ data structure.

**Algorithm 1:** Partitioning graph $G$ using a partitioning scheme $(P, P_{\text{root}})$

```
Procedure partition($G, P, P_{\text{root}}$)
  for each path $\delta$ in $\text{paths}(P, P_{\text{root}})$ do
    eval($G, \delta$)
  end
Procedure eval($G, \delta = x, e_1, o_1, \ldots, e_n, o_n$)
  for each triple $(s, p, o)$ in $G$ do
    if $(s, p, o)$ matches $(x, e_1, o_1)$ then
      $G_s \leftarrow (s, p, o)$
      $T_1 \leftarrow (s, (s, p, o))$
    end
  end
  for $i = 2 \ldots n$ do
    for each triple $t$ in $G$ do
      if $t$ matches $(o_{i-1}, e_i, o_i)$ and joins with $(s, t')$ in $T_{i-1}$ according to $\delta$ then
        $G_s \leftarrow t$
        $T_i \leftarrow (s, t)$
      end
    end
  end
```

Notice that a triple $t$ is added to a partition $G_s$ if and only if $t = (s, p, o)$ matches $(P_{\text{root}}, e_1, o_1)$ (line 8) or $t = (x, y, z)$ is part of a path $s, p, o, \ldots, x, y, z$ that matches a path $\delta = P_{\text{root}}, e_1, o_1, \ldots, e_m, o_m$ that is in $\text{paths}(P, P_{\text{root}})$ (lines 12-18). The data
### Table 7.1: The result of evaluating the path \(?x, \text{friend}, \ ?z, \ ^\text{friend}, \ ?y\) against the graph of Figure 7.1 using Algorithm 1 (\text{friend} is abbreviated as \(\text{:fr}\)). The first column matches \(P_{\text{root}}\) and determines, therefore, the subgraph to which the triples will be added.

<table>
<thead>
<tr>
<th></th>
<th>(T_1)</th>
<th>(T_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sarah</td>
<td>([??]_G)</td>
<td>Sarah :fr Ronan</td>
</tr>
<tr>
<td>Peter</td>
<td>Peter :fr Ronan</td>
<td></td>
</tr>
<tr>
<td>Ronan</td>
<td>John :fr Ronan</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td>John :fr Peter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>John :fr Sarah</td>
<td></td>
</tr>
</tbody>
</table>

Structure \(T_i\) ensures that the node that was mapped to \(P_{\text{root}}\) is saved and used to route all matching triples. Algorithm 1 is a generalisation of the one used in [81] for \(n\)-hops partitioning. In contrast to the algorithm reported in [81], Algorithm 1 considers paths defined in a given pattern and adds only matching triples.

For a pattern \(P\) with \(n\) triple patterns, the algorithm reads the data \(n\) times and performs \(n - 1\) joins. The main two operations in the algorithm (i.e., filtering and joining) makes it easy to be implemented in popular distributed computation platforms such as Apache Spark or MapReduce. The Optimised repartition join algorithm [80] is a commonly used algorithm for implementing join operation in MapReduce platform. However, for selective patterns, more efficient join algorithms in a shared-nothing clusters can be performed by broadcasting the smaller dataset for instance.
Chapter 7. Pattern-based Graph Partitioning

7.2.3 Data Placement

The next step in graph partitioning is assigning each subgraph to a partition\(^6\). This assignment should aim at achieving two goals; (i) ensuring balanced storage across partitions (i.e., partitions are of similar sizes) and (ii) minimising duplication among partitions. For instance, one possible assignment of subgraphs shown in Figure 7.3 to two partitions is to assign subgraphs \(G_{Peter}\) and \(G_{Ronan}\) to one partition and subgraphs \(G_{Sarah}\) and \(G_{John}\) to the other (i.e., in the figure, subgraphs \(A\) and \(B\) form the first partition while subgraphs \(C\) and \(D\) form the second partition). The first partition will have 11 triples in total and the second will have 12. Alternatively, assigning subgraphs \(G_{Peter}\) and \(G_{Sarah}\) to one partition and subgraphs \(G_{Ronan}\) and \(G_{John}\) to the other partition results in two partitions of 9 and 12 triples, respectively. The second assignment achieves more reduction of duplicate because it assigns subgraphs that share more triples to the same partition.

Similar to [88], we cluster subgraphs based on their shared triples before assigning them to partitions. We use the K-medoids algorithm [84] for the clustering where \(k\) is the number of intended partitions. The clustering algorithm should also track the size of each cluster and ensure that clusters are of similar sizes. We

\(^6\)One can think of partitions and machines in a cluster as being equivalent in this context.
used Jaccard similarity to measure similarity between subgraphs. Nevertheless, calculating the similarity between the actual subgraphs can be costly. The number of triples in the subgraphs is potentially large and they are distributed across multiple machines. The communication and computation cost of clustering would be very high. Therefore, we use a bitmap index as an approximate representation of a set of triples that constitutes each of the subgraphs. A combination of logical bitwise and and or operators is used to calculate the distances between bitmaps. The size of the bitmap indices can be adjusted to control the possibility of false positives. We report the effect of clustering on data placement in our experimental evaluation in the next chapter.

The K-medoids algorithm has a number of appealing features for our needs. These include [127]: (i) it can work with arbitrary metrics of distances between data-points, (ii) it is more robust to noise and outliers as compared to k-means, (iii) the algorithm takes the number of clusters as an input, and (iv) it can be implemented using a fast greedy algorithm. On the other hand, it should be noted that K-medoids algorithm may converge to a local optimum and may not find the optimum solution. There exists a number of different clustering algorithms that can be used and a full study of them is beyond the scope of this thesis.

7.3 Query Answering

As the data is partitioned across different machines, it is commonly the case that no single machine has the full answer to a given query $Q$. Therefore, it is necessary to decompose the query $Q$ into sub-queries that can be fully answered by each machine and then combine the results of the subqueries to build the final answer to $Q$. A naive decomposition of $Q$ into individual triple patterns ensures a correct result. However, such decomposition introduces a high cost as it results into a high number of joins (if $Q$ contains $n$ triple patterns, $n - 1$ joins are needed). Therefore, the problem can be stated as follows: given a query $Q$ and a graph $G$ partitioned according to a partitioning scheme $(P, P_{root})$, find the optimal decomposition of $Q$ that minimises the evaluation time.
In distributed systems, the communication over the network\textsuperscript{7} is an important factor to optimise. In the absence of statistics to estimate the cost of different joins, we resort to the following heuristic: the smaller the number of joins is, the less data communication is needed over the network. This heuristic is similar to the one used in [81, 99, 171]. Therefore, the goal is to decompose the query $Q$ into the smallest number of subqueries each of which can be fully answered intra-partition (i.e., with no communication across partitions).

As a prerequisite, we need to determine whether a query $q$ can be answered intra-partition given a partitioning scheme $(P, P_{\text{root}})$. Because a partition is defined using a graph pattern which is also a query, this task is basically about the relationship between two queries. Particularly, the task is to determine whether the result of a query $P$ is sufficient to answer a given query $q$. This problem is related to view-based query answering. View-based query answering has been studied in relational databases [32], XML [119], and description logic [14]. The peculiarities of our problem is that: (i) the semantics of partitioning and replication is different than that of query answering. Partitioning pattern can be interpreted as a union of multiple select-project-join queries (i.e., partial solutions to the partitioning pattern are still added to corresponding partitions), (ii) partitions (corresponding to views) are rooted and distributed, and (iii) partitions are defined using graph patterns.

In order to reason about queries, we reuse the partial order relationship between RDF terms that we defined in Chapter 4 (repeated here in Definition 7.6). We generalise the relationship to paths and then present sufficient conditions to determine the answerability of a query, given a partitioning scheme.

**Definition 7.6.** $\forall x_1, x_2 \in T \cup V : x_1 \preceq x_2$ iff one of the following holds:

- Both $x_1$ and $x_2$ are variables (i.e., $x_1, x_2 \in V$).
- $x_1$ and $x_2$ are equal RDF terms (i.e., $x_1, x_2 \in T \land x_1 = x_2$).
- $x_1$ is a term and $x_2$ is a variable (i.e., $x_1 \in T \land x_2 \in V$).

We generalise $\preceq$ to order paths.

\textsuperscript{7}For example, the cost model for distributed computation described in [7] uses the size of data transferred over the network as the main measure.
**Definition 7.7.** For two paths \( \delta = o_0, e_1, o_1, \ldots, e_n, o_n \) and \( \gamma = f_0, r_1, f_1, \ldots, r_n, f_n \) we say that \( \delta \preceq \gamma \) iff:

1. For \( 0 \leq i \leq n \): \( o_i \preceq f_i \)
2. For \( 1 \leq i \leq n \); one of the following holds:
   - \( e_i \in I \cup V, r_i \in I \cup V \) and \( e_i \preceq r_i \)
   - \( e_i \) is of the form \( \hat{p}_i \), \( r_i \) is of the form \( \hat{q}_i \) and \( p_i \preceq q_i \)

**Theorem 7.1.** Given a query \( Q \) and a graph \( G \) partitioned according to a partitioning scheme \( (P, P_{\text{root}}) \) using Algorithm 1, \( Q \) can be answered intra-partition if it can be decomposed into a set of paths \( \{\delta_1, \delta_2, \ldots, \delta_m\} \) all rooted in the same node \( s \); such that there exists a mapping \( \sigma: \{\delta_1, \delta_2, \ldots, \delta_m\} \rightarrow \text{paths}(P, P_{\text{root}}) \) where \( \delta_i \preceq \sigma(\delta_i) \) for \( 1 \leq i \leq m \).

**Proof.** We show that, given the existence of a mapping \( \sigma \) as required by the theorem, for any two triples \( t_1, t_2 \) that are used in a result of evaluating \( Q \) against \( G \), it is always the case that \( t_1 \) and \( t_2 \) are located together in some partition.

We distinguish two cases:

**Case 1.** \( t_1 \) and \( t_2 \) are in the result of matching some path \( \delta_i = o_0, e_1, o_1, \ldots, e_n, o_n \) for some \( 1 \leq i \leq m \) (notice that \( o_0 = s \) as all paths \( \delta_i \) are rooted at \( s \)). Therefore, for some \( k; j \in [0, n - 1] \) it is the case that \( t_1 \preceq (o_k, e_{k+1}, o_{k+1}) \) and \( t_2 \preceq (o_j, e_{j+1}, o_{j+1}) \) based on the definition of \( \preceq \). If \( \sigma(\delta_i) = f_0, r_1, f_1, \ldots, r_n, f_n \), then \( o_0, e_1, o_1, \ldots, e_n, o_n \preceq f_0, r_1, f_1, \ldots, r_n, f_n \) as required in the theorem. Therefore, \( t_1 \preceq (f_k, r_{k+1}, f_{k+1}) \) and \( t_2 \preceq (f_j, r_{j+1}, f_{j+1}) \) because \( \preceq \) is transitive. Consequently, \( t_1 \) and \( t_2 \) are part of matching the path \( \sigma(\delta_i) \) against \( G \) and are, therefore, assigned to the same partition by Algorithm 1 because they are matched by a path in \( \text{paths}(P, P_{\text{root}}) \).

**Case 2.** \( t_1 \) and \( t_2 \) are matched by two paths \( \delta_i \) and \( \delta_j \), respectively. Similar to the discussion in case 1, we can infer that \( t_1 \) and \( t_2 \) are matched by \( \sigma(\delta_i) \) and \( \sigma(\delta_j) \), respectively. \( \delta_i \) and \( \delta_j \) share the same root \( s \). Moreover, \( \sigma(\delta_i) \) and \( \sigma(\delta_j) \) also share the same root, \( P_{\text{root}} \) (because they are in \( \text{paths}(P, P_{\text{root}}) \)). As a result, \( t_1 \) and \( t_2 \) are assigned to the same partition by Algorithm 1 because they are matched by two paths that share the root.
Consequently, the existence of a mapping $\sigma$ that satisfies the conditions of the theorem guarantees that the query $Q$ can be answered with no communications across partitions.

Figure 7.4 shows a possible decomposition and a mapping function of the query and the pattern shown previously in Figure 7.2.

We believe that Theorem 7.1 can be useful in other scenarios when there is a need to reason about a query answerability given the answer of another query. For example, Linked Data Fragments is recent proposal to scale Linked Data querying [167]. Work on Linked Data Fragments has been restricted so far to simple individual triple patterns. Theorem 7.1 can provide a basis to apply fragments to more complex graph patterns. Theorem 7.1 might also be useful in reasoning about the applicability of cached RDF query results given some query.

### 7.4 Related Work

As shown in the previous section, the problem of query decomposition was reduced to that of view-based query answering. We discuss here work related to view-based query answering, while work related to RDF partitioning on distributed frameworks is discussed after describing our implementation in the next chapter. View-based query answering is based on query containment and has been extensively studied in literature. Original work on containment of select-project-join relational queries was presented by Chandra et al. in a seminal paper in 1977 [27]. Chandra’s paper also shows that the problem of deciding on query
containment is NP-complete. Later results extended Chandra’s work to support predicates [102, 103]. [32] provides an algorithm for optimising queries in the presence of materialised views and [69] provides a comprehensive survey.

Our requirements for answerability are similar to that presented in [27, 32, 102]. However our notions of paths and the ≤ relation capture the graph nature of the data and the query more intuitively. For instance, using the criteria defined in [102], to decide that $Q_1$ contains $Q_2$ we need to find a decomposition of $Q_2$ into a number of queries $Q_2^1, ..., Q_2^n$ such that:

- $Q_2^1, ..., Q_2^n$ differ only in their built-in literals, and
- $Q_2$ is equivalent to the union of $Q_2^1, ..., Q_2^n$, and
- For every $i$, $1 \leq i \leq n$, there is a containment mapping $\phi_i$ from $Q_1$ to $Q_2^i$, such that $b_i(Q_2^i)$ entails $\phi_i(b_i(Q_1))$ where $b_i(Q)$ is the conjunction of built-in atoms in the query $Q$.

Our criteria in Theorem 7.1 replaces the query decomposition into paths decomposition and utilises the facts that views (aka partitions) are rooted and that all predicates involved are equality predicates.

SPARQL queries are more complicated than select-project-join queries and, therefore, deciding on query containment for SPARQL is more challenging. [101] studied query containment for SPARQL queries with OPTIONAL clauses, while [132] studied also union and projection queries. The complexity of deciding query containment for different fragments of SPARQL ranges from NP-completeness to undecidability [101, 132]. Our work discusses only select-project-join queries, leaving more complex queries for a future work. The containment of select-project-join queries is known to be NP-complete [5].

### 7.5 Summary

This chapter presented pattern-based partitioning for RDF data. Algorithms necessary to partition an RDF graph, distribute the partitions across machines, and answer queries using the partitioned data were presented in this chapter. In the next chapter, we detail an implementation and an experimental evaluation of these algorithms.
Chapter 8

Implementation and Experimental Study

In this chapter, we describe an implementation and experimental evaluation of the algorithms described in the previous chapter.

8.1 Implementation

Our implementation uses Java and the new Hadoop 2.0 API\(^1\). The overall architecture of our implementation is shown in Figure 8.1. The query decomposer breaks up a given query into a set of subqueries that all can be answered intra-partition as described in Section 7.3. Given the small number of triple patterns in typical SPARQL queries, query decomposition is realised through a brute-force searching method. Then the query executor sends each subquery to each node where the query gets evaluated as a MapReduce map-only job. Similar to [81] and [99], a triple store manages the data locally at each of the cluster nodes. Query results provided by the triple stores are written to the Hadoop distributed file system (HDFS). Afterwards, the query executor joins results of subqueries to compute the full query results. Joining these subresults is coordinated by Hadoop using the optimised repartition join algorithm [80]. For data placement, similar subgraphs are clustered together in order to reduce replication. We used an implementation of the K-medoids algorithm that produces similar-size clusters, as described in [2].

\(^1\)http://hadoop.apache.org/
8.2 Experimental Study

For experimental evaluation, two sets of experiments were conducted. Synthetic data using the Lehigh University Benchmark (LUBM) [67] was used in one, whereas real-world RDF statistical data was used in the other. In this section, we describe the settings of each of the two experiments and report the results.

8.2.1 Experiment Setup

LUBM Experiment

We generated data about 4000 universities and materialised inferenced triples based on the LUBM ontology using basic RDFS reasoning. The result dataset contains 650 million triples which add up to a total size of 110 GB when serialised as N-Triples. The experiment was run on the Amazon Cloud Computing Service. A cluster of 15 machines was used. Each machine is of type Amazon m3.xlarge.
(4 virtual CPU cores and 15 GB RAM). We used Apache Hadoop version 2.2.0 running on Java 1.6.1. For triple stores, we used RDF-3X 0.3.8.

We implemented and evaluated four partitioning patterns of different levels of complexity:

**Star** The star-shaped graph partitioning (aka directed 1-hop guarantee). The partitioning scheme is \( \{(s, p, o), (s)\} \). This is used as a baseline to compare the other partitioning schemes.

**2-hop** The directed 2-hop guarantee partitioning. The partitioning scheme is \( \{(s, p, o), (o, p2, o2), (s)\} \).

**rdfType** The partitioning scheme is \( \{(s, p, o), (o, \text{rdf:type}, t), (s)\} \). Notice that RDFS inference performed on the LUBM data adds a large number of \text{rdf:type} statements. Therefore, this partitioning pattern is used as an example of a non-selective pattern.

**subOrg** The partitioning scheme is \( \{(s, p, o), (o, \text{ub:subOrganizationOf}, v), (s)\} \). This partitioning pattern is used as a selective pattern as the number of triples with property \text{ub:subOrganizationOf} is not large in the data.

**Statistical Data Experiment**

We are interested in querying for statistics across different statistical datasets. In this experiment, we used the RDF version of the World Bank statistics and part of the European Central Bank Linked Data\(^2\). Both datasets use the Data Cube vocabulary\(^3\) sketched in Figure 8.2. The two datasets are interlinked using \text{skos:exactMatch} property. The size of the data used is 230 Million triples adding up to a 55 GB when serialised as N-Triples.

Our experiments were run on a 10-node cluster running on the Google Compute Engine\(^4\). Each node is of type \text{Google n1-standard-2} (8 GB RAM, 2 virtual CPU cores)\(^5\). We used Apache Hadoop version 2.2.0 running on Java 1.6.1. For triple stores, we used RDF-3X 0.3.8.

---

\(^2\)http://worldbank.270a.info and http://ecb.270a.info

\(^3\)http://www.w3.org/TR/vocab-data-cube

\(^4\)https://cloud.google.com/products/compute-engine/

\(^5\)An image of the machines used is available at: http://bit.ly/1rZgxiN
In addition to the star and 2-hop partitioning schemes, we used the partitioning scheme \texttt{skosMatch}: \texttt{\{\{s, ?p, o\}, \{o, skos:exactMatch, v\}\}}, \texttt{s}).

### 8.2.2 Results and Discussion

**Data Partitioning and Placement**

Table 8.1 and Table 8.2 show the loading time and the duplicated triples added using the different schemes. Notice that, in the LUBM experiment, \texttt{2-hop} adds 1.8 billion triples as the total size tends to grow exponentially as the graph is navigated. \texttt{rdfType} also introduces a significant overhead. In the statistical data we used, any two observations in a dataset are no more than two hops away from each other via a \texttt{cube:dataSet, rdfs:seeAlso} path (see Figure 8.2). Consequently, our initial attempt to run the 2-hop partitioning of the data failed as the size of replicated data was growing very fast to the point that it overwhelmed the network. Therefore, \texttt{rdfs:seeAlso} links were removed for the 2-hop partitioning scheme. Nevertheless, the total size of the data when using \texttt{2-hop} became 120 GB (approximately, a 218% increase in size).

It is worth pointing out that the number of actual triples that are saved and processed gets smaller when the data is loaded in each node as each node removes duplicated triples locally. The clustering we used during data placement is meant to maximise reduction in duplicates. Figure 8.3 and table 8.3 present the effect of clustering.

The extent to which clustering manages to reduce duplication was little in general. This might be due to the approximation used during clustering through bitmaps.
### Table 8.1: LUBM - Loading time and data redundancy

<table>
<thead>
<tr>
<th></th>
<th>#duplicate triples (millions)</th>
<th>Loading time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star</td>
<td>0 (0%)</td>
<td>18m 49s</td>
</tr>
<tr>
<td>2-hop</td>
<td>1800 (277%)</td>
<td>78m 56s</td>
</tr>
<tr>
<td>rdfType</td>
<td>830 (127%)</td>
<td>54m 16s</td>
</tr>
<tr>
<td>subOrg</td>
<td>88 (14%)</td>
<td>32m 19s</td>
</tr>
</tbody>
</table>

### Table 8.2: Statistical data - Loading time and data redundancy

<table>
<thead>
<tr>
<th></th>
<th>Data per node (GB)</th>
<th>Loading time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star</td>
<td>5.5</td>
<td>27</td>
</tr>
<tr>
<td>2-hop</td>
<td>12</td>
<td>87.83</td>
</tr>
<tr>
<td>skosMatch</td>
<td>6</td>
<td>48.35</td>
</tr>
</tbody>
</table>

### Figure 8.3: LUBM - Distribution of partitions sizes (small horizontal line is the mean of partitions sizes)
### Table 8.3: LUBM - Comparison of the size of data between random placement (-rand) and clustering (-clust)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#total triples (millions)</th>
<th>Avg. #triples per node (millions)</th>
<th>range of #triples per node (max - min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-hop-clust</td>
<td>2024</td>
<td>135</td>
<td>41.5</td>
</tr>
<tr>
<td>2-hop-rand</td>
<td>2240.1</td>
<td>149.3</td>
<td>32.2</td>
</tr>
<tr>
<td>rdfType-clust</td>
<td>1351.9</td>
<td>90</td>
<td>31.5</td>
</tr>
<tr>
<td>rdfType-rand</td>
<td>1389.3</td>
<td>92.6</td>
<td>31</td>
</tr>
<tr>
<td>subOrg-clust</td>
<td>690.6</td>
<td>46</td>
<td>7.9</td>
</tr>
<tr>
<td>subOrg-rand</td>
<td>697.5</td>
<td>46.5</td>
<td>9.2</td>
</tr>
</tbody>
</table>

and the used distance measure (i.e., the Jaccard similarity). Jaccard similarity between two sets counts the number of common elements between the two sets. Therefore, when applied to two sets of RDF resources, Jaccard similarity counts the number of common resources regardless of the number of triples these common RDF resources have. A similarity measure that considers nodes degrees in the graph (i.e., the number of triples an RDF resource is part of) might prove more fruitful. Further work is required to better understand the effect of clustering. However, in general clustering reduces reduction to some extent when the pattern is non-selective but has little effect otherwise. Partitions were of similar sizes which indicates that the load balancing guard utilised was effective.

**Data Querying**

The LUBM benchmark contains fourteen queries representing different goals and various levels of complexity. Eight of the queries are simple star queries. Star queries, expectedly, showed no distinction in query response time between the different partitioning schemes as they do not need to be decomposed under any of the used schemes. Therefore, we report only the remaining six queries (Table 8.4).

For statistical data, we used five SPARQL queries that search for related items and related datasets across the two studied datasets. The SPARQL queries used are reported in Appendix A and their decomposition under the different partitioning schemes are shown in Table 8.5.

Response times for LUBM and statistical data queries are shown in Figure 8.4 and Figure 8.5, respectively. Performance of the different partitioning patterns
with respect to each query is plotted. The figures need to be interpreted based on Table 8.4 and Table 8.5 that show the decomposition of the queries under each of the partitioning schemes. The number of joins required is one less than the number of subqueries. In general, the query response time is dominated by the join time. Local triple stores responded fast to individual subqueries. In Figure 8.4, the 2-hop and rdfType schemes answered all the queries in less than 10 minutes. The subOrg scheme does not match all the queries in less than 10 minutes.

In Figure 8.5, it is worth pointing out that query 2 failed for the star partitioning due to non-selective patterns in the query that resulted in a huge amount of data shuffling. Generally, the tailored partitioning scheme, skosMatch, showed response times competitive to those of the 2-hop except for query 4 which does not include a skos:exactMatch pattern.
In conclusion, using patterns enables control over partitioning and replication of RDF data. Star and 2-hop methods, for instance, represent different trade-offs where the former maintains the low loading time and supports simpler queries well, whereas the latter supports both simple and complex queries at the cost of the increase in the load time and data replication. On the other hand, more selective patterns can refine partitioning to better suit the data and show good performance when they match the anticipated query patterns. This confirms the motivation for using pattern-based graph partitioning. Nevertheless, it is not possible to provide a general recommendation of which partitioning patterns to use. Such decision is dependant on the data and the query load.
Chapter 8. Pattern-based Graph Partitioning

8.3 Related Work

Related literature describes a significant number of SPARQL Query processing systems built on top of MapReduce (\cite{60} provides a recent survey). When it comes to partitioning the data, these systems adopt various methods. The default random hashing is used in SHARD \cite{139} and in few other systems \cite{59, 122}. PigSparql \cite{143} and Rapid+ \cite{138} are built on top of Apache Pig\footnote{https://pig.apache.org/}. As Pig is oblivious to the nature of the data, it uses the default random hashing. Other systems partition the RDF triples based on their subject, predicate, object, or a combination of them. CliqueSquare \cite{62}, for instance, partitions the data based on subject and then partition the data on each node based on their predicate. Such intra-node partitioning is a local indexing that complements the global distributed index implied by partitioning the data across nodes.

Beyond hashing based on a single triple component, graph partitioning algorithms are used in some recent work \cite{81, 99}. For example, \cite{81} uses the min-cut graph algorithm for the partitioning. Our work is also a graph partitioning method but we support selective replication instead of the exhaustive approach used in both \cite{99} and \cite{81}. These two systems deploy a triple store on each node to index the data locally in the same way we used in our implementation.

Outside the RDF world, CoHadoop \cite{52} provides a lightweight extension to Hadoop which is designed to enable co-locating related files at the file system level. We share the goal of increasing data locality with CoHadoop. Nevertheless, our work focuses on RDF data and considers a single triple as the basic building block while CoHadoop works at the file system level.

Anticipating query load to enhance data layout and partitioning is a well-known technique in relational database and it was also utilised in recent Hadoop work such as HadoopDB \cite{6} and Hadoop++ \cite{51}.

8.4 Limitations

Our work on partitioning focuses on domain-specific graphs as reflected by the graphs used in the experimental evaluations. Such domain-specific graphs have
recurring patterns and simple underlying ontologies that make it easier to meaningfully choose a partitioning pattern. Furthermore, we only showed algorithms and implementation for realising star-shaped patterns that are enriched with further replicated triples.

8.5 Summary

This chapter presented an implementation of pattern-based graph partitioning. An experimental study, using both synthetic and real-world data, was also described.

This concludes our work on using graph patterns to guide graph partitioning. This part detailed a formal model, pertinent algorithms, an implementation, and an experimental study of the data storage layer of our work. In the last part, we describe a data-flow language that presents the user facing interface to interact with the data model and the storage layer presented in the first two parts of this thesis.
Part III

SYRql, the Language Layer
Chapter 9

SYRql, a Dataflow Language for RDF

In this chapter, we describe SYRql, a dataflow language for large scale RDF processing. SYRql language is designed on top of the RDF.co data model described in Chapter 3. We discuss utilising RDF.co properties (Chapter 4) and pattern-based graph partitioning (Chapter 6) in order to optimise the evaluation of a SYRql script in Section 9.2. Our implementation of the SYRql language is described in Section 9.3. An experimental study comparing SYRql to other existing big data languages is reported in Section 9.4.

9.1 SYRql, a Dataflow Language for RDF

SYRql is a dataflow language that is grounded in the RDF.co data model defined in Chapter 3. The core set of operators in SYRql are those defined in Section 3.1.2, namely: bAccess, PROJ, bEXT, gEXT, MATCH, FILTER, and CROSS. Moreover, SYRql also supports the aggregation operators defined in Chapter 5. A number of the derived operators such as gAccess and GMATCH are also supported in SYRql. A SYRql program is a sequence of statements and each statement is either an assignment or an expression. Therefore, a SYRql program is conceptually a directed graph where nodes are operations and edges represent data flowing between the operators.
SYRql provides the user with the syntax to express dataflow programs based on RDF.co operators. Listing 9.1 shows a part of the BNF grammar defining the syntax of SYRql. The syntax of SYRql borrows the use of “→” syntax from Jaql [16] to explicitly show the data flow. According to the designers of Jaql, the “→” syntax, inspired by the Unix pipes, makes scripts easier to read and debug [16]. It allows eliminating the need for defining variables (as in Pig Latin [125]) or for a WITH clause (as in SQL) in each computational step. It is worth mentioning that the Meteor [77] language dropped the pipe notation of Jaql to support operators with multiple inputs and outputs. In SYRql, operators with multiple inputs or outputs are not common and therefore we decided to adopt the pipe syntax. However, SYRql does support multi-input operators such as multi-way joins.

Pattern matching in SYRql uses identical syntax to basic graph patterns of SPARQL. SPARQL syntax for patterns is intuitive, concise and well-known to many users in the Semantic Web field. We hope that this facilitates learning SYRql for many users. Notice that graph patterns are equivalent to \texttt{GMATCH} operator defined as part of RDF.co.

```plaintext
| exprList  | (expr ';'*) |
| expr      | expr '→' opCall |
|           | ID '=' expr |
|           | funCall |
|           | ID |
|           | constant |
| opCall    | rdfPattern |
|           | bAccess |
|           | gAccess |
|           | joinOp |
|           | filterOp |
|           | crossOp |
|           | smplFunCall |
|           | groupByOp |
| groupByOp | 'group' 'by' varList 'into' fieldList |
| filterOp  | 'filter' '(' logicalExp ')' |
| crossOp   | 'cross' expr |
| funCall   | joinExpr |
|           | smplFunCall |
| joinOp    | 'join' expr |
```
Chapter 9. **SYRql**

Listing 9.1: Part of the Syntax Grammar of SYRql

<table>
<thead>
<tr>
<th>joinExpr</th>
<th>':join expr ', expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdfPattern</td>
<td>'pattern ' '(' ' triples ' ')'</td>
</tr>
<tr>
<td>smplFunCall</td>
<td>'ID '(' argList ')'</td>
</tr>
</tbody>
</table>
| COMMENT        | '// .s? ('
'|EOF) |

Listing 9.2 shows an example SYRql program that performs pattern matching, filtering, and aggregation. Notably, line 10 in the script provides an example of composability that is not directly available in SPARQL. In line 10, a pattern matching is applied to the results of another pattern matching. We believe that such capabilities are useful for complicated tasks, specifically for exploratory tasks, and for reusing previous scripts as well as previously computed results. Further description and examples of the SYRql language is available online\(^1\). The full BNF grammar defining the syntax can also be found on the SYRql website.

```plaintext
$rdf = load('hdfs://master:9001/bsbm20k');

$janReviewers = $rdf -> pattern('?review rev:reviewer ?reviewer .
  ?review dc:date ?date .
  ?reviewer bsbm:country ?cntry .')
  -> filter(?date >= '2008-01-01' && ?date < '2008-02-01');

$janReviewers -> group by ?cntry into janReviewersCount:count(?review);

$irelandJanReviewers = $janReviewers -> pattern('?rev bsbm:country :IE');
```

Listing 9.2: Example SYRql program

### 9.2 Execution and Optimisation of SYRql Programs

The main goal of SYRql is to execute dataflow programs on top of big data sets and, therefore, SYRql programs are designed to run in parallel. To enable parallel operations to run efficiently, SYRql adopts a lazy evaluation model. The use of a SYRql operator does not directly execute the operator, but instead simply records the operation and its arguments in an internal execution plan graph structure.

\(^1\)https://gitlab.insight-centre.org/Maali/syrql-jsonld-imp/wikis/home
Chapter 9. SYRql

This model was also adopted in Pig Latin [125], Apache Spark [1], and Flume-Java [26]. A number of optimisations are applied to the execution plan before it is actually run. We discuss the optimisations that can be applied to a SYRql script. Details of our implementation is described in the next section.

Data Layout Partitioning the data using a graph pattern as defined in Chapter 7 can be utilised while evaluating MATCH and GMATCH operators. The given query can be decomposed into a set of subqueries in order to minimise the required number of joins, and therefore to minimise the communication over the network.

Coalescing Operators Consecutive operators that do not require shuffling the data around can be coalesced together and run as a single job. In other words, some operators can be pipelined together to allow the flow of the results from one to another without the need to fully materialise intermediary results. For example, two consecutive pattern matching or a pattern matching and a filter operator can be run in a single scan of the data. In RDF.co, operators that require data shuffling are CROSS and AGG. Therefore, these two operators define the grouping of a chain of SYRql operators into jobs.

RDF.co Rewriting Rules Chapter 4 defines a number of algebraic properties of RDF.co data model. These properties can be used as rewriting rules to achieve a more efficient execution plan for running a SYRql script. In particular, two sets of properties were defined and can be used: cascading triple pattern matching (Section 4.1); and evaluating graph and binding components of expressions (Section 4.2).

9.3 SYRql Implementation

SYRql and RDF.co define a collection of operations that provide a high-level uniform abstraction that can be implemented over different data representations and distributed systems. The current implementation\(^2\) translates SYRql scripts into a series of MapReduce [46] jobs. We use Java and Apache Hadoop 2 API\(^3\) in our implementation.

\(^2\)https://gitlab.insight-centre.org/Maali/syrql-jsonld-imp
\(^3\)http://hadoop.apache.org/
9.3.1 Data Representation

JSON\(^4\) is used for internal representation of the data. Particularly, we use JSON arrays for bindings and JSON-LD [149] to represent graphs. JSON-LD is a recent W3C recommended serialisation of RDF. It has attracted a good adoption so far and this can be expected to grow. Consequently, by using JSON-LD a large amount of RDF data can be directly processed using SYRql. Furthermore, we utilise JSON-LD’s ability to represent star subgraphs as single JSON objects, thus eliminating the need for joins when evaluating star-join queries. This particular way of encoding RDF in JSON-LD is referred to as the flattened document form\(^5\) and is the format used in our SYRql implementation. Moreover, we provide a MapReduce implementation that converts RDF data serialised as N-Triple format\(^6\) into flattened JSON-LD.

9.3.2 Parsing, Compiling and Execution

Figure 9.1 shows an overview of the main components involved in executing a SYRql program. We use ANTLR\(^7\) to parse SYRql scripts and build the abstract syntax tree. Each node in the tree represents an expression and the children of the node are its inputs. For triple matching expressions, triple patterns are grouped by subject to utilise the data locality provided by partitioning the data as star-structured subgraphs, thus reducing the number of required joins. A number of optimisations, discussed in the next section, is applied to the tree to reduce the expected execution time. The optimised tree is then translated into a directed acyclic graph (DAG) of MapReduce jobs. Sequences of expressions that can be evaluated together are grouped into a single MapReduce job (i.e., operators coalescing). Finally, the graph is topologically sorted and the MapReduce jobs are scheduled to execute on the cluster. It is worth mentioning that for join expressions we implemented the optimised repartition join algorithm [80].

\(^4\)http://json.org  
\(^5\)http://www.w3.org/TR/json-ld/#flattened-document-form  
\(^6\)http://www.w3.org/TR/2014/REC-n-triples-20140225/  
\(^7\)http://www.antlr.org/
9.3.3 Implemented Optimisations

In our implementation, we used a simple star pattern to partition the data (i.e., the partitioning scheme used is \( \{(s, p, o), (s)\} \)). Partitioning by subject is a cost effective method that covers star-shaped query loads in isolation from the domain and the characteristics of the datasets. Configuring data partitioning to use other more complex patterns will be added in future implementation. Furthermore, we implemented coalescing consecutive operators.

Of the algebraic properties defined in Chapter 4, we implemented only the rewriting rules defined in Section 4.2 in order to reduce the cost of evaluating both the graph and the binding components of SYRql expressions.

For example, the short SYRql script:

\[
e_1 \rightarrow \text{cross } e_2 \rightarrow \text{pattern}(s \ p \ ?o) \rightarrow \text{filter}(?o = 'a') \rightarrow b()
\]

is translated into the following RDF.co expression:

\[
((e_1 \times e_2)[(s, p, ?)][1 = "a"]\).b
\]

Figure 9.2 shows a possible naive execution plan of the script. Notice that the naive plan requires performing a costly \texttt{CROSS} operator on both the bindings and the graph components of \( e_1 \) and \( e_2 \). On the other hand, an optimised plan is shown in Figure 9.3. Notice that the optimised plan applies the \texttt{CROSS} operator only on the graph components of the involved expressions. This can significantly reduce the cost of the evaluation. Additionally, the optimised plan coalesces the \texttt{MATCH} and the \texttt{FILTER} operators in a single map-only job. The optimised plan is achieved by utilising the properties of the involved operators. Namely, the facts that \texttt{bAccess} is g-independent, \texttt{CROSS} is component-separable, and \texttt{MATCH} is b-independent.
9.4 Evaluation

We conducted a performance evaluation of SYRql implementation. Our goal of this evaluation is two-fold:

- Compare performance of SYRql to other popular alternatives, namely Jaql, Pig Latin, and HiveQL. Our thesis is that SYRql’s features and syntax that are specifically designed for dataflow programs on top of RDF data can help generating scripts that are easier to understand and debug. Therefore, we want to ensure that our implementation achieves similar level of performance to match existing big data languages.
Figure 9.3: An optimised execution plan for the expression \(((e_1 \times e_2)[(s,p,?)][1 = "a"]).b\)

- In the same spirit of Pig Mix\(^8\) that is developed as part of Pig Latin, we want this benchmark to measure performance on a regular basis so that the effects of individual code changes on performance could be understood.

We based our benchmark on the Berlin SPARQL Benchmark (BSBM) \(^{18}\) that defines an e-commerce use case. Specifically, we translated a number of queries in the BSBM Business Intelligence usecase (BSBM BI)\(^9\) into equivalent programs in a number of popular big data languages. BSBM BI queries exercise aggregate functions and provide analytical queries that involve bigger portion of the data to get the required results. In this regards, the BSBM BI benchmark matches

\(^8\)https://cwiki.apache.org/confluence/display/PIG/PigMix
\(^9\)http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BusinessIntelligenceUseCase/index.html
the requirements presented in Section 2.5.5. In particular, we provide programs equivalent to the queries in the following languages:

**Jaql**  A scripting language designed for Javascript Object Notation (JSON).

**Pig Latin**  A dataflow language that provides high-level data manipulation constructs that are similar to relational algebra operators.

**HiveQL**  A declarative language that uses a SQL syntax.

The scripts used were customised to utilise the strong features of each of the used languages. We believe that they reflect what an interested user would write given a reasonable amount of time. We evaluated four queries from BSBM BI that cover all core operators i.e., filters, patterns, joins and aggregation.

![Figure 9.4: Query processing times](image)

**9.4.1 Experimental Setup**

**Environment:** The experiments were conducted on VCL\(^\text{10}\), an on-demand computing and service-oriented technology that provides remote access to virtualised resources. Nodes in the clusters had minimum specifications of single or duo core Intel X86 machines with 2.33 GHz processor speed, 4GB memory and running Red Hat Linux. We used a 10-node cluster and the following software versions: Apache Hadoop 2.3.0, Jaql 0.5.1, Pig 0.12.1, and Hive 0.12.0. Apart from setting memory per machine and the number of reducers, default configurations for each language were used. For Hive, we used ORC files to store the data.

\(^\text{10}\) [https://vcl.ncsu.edu/](https://vcl.ncsu.edu/)
Dataset and queries: We generated BSBM data for 400K products in N-triple format. The size of the data was about 35GB containing approximately 140 million triples. As mentioned before, the queries are the scripts corresponding to BSBM BI queries.

9.4.2 Results & Discussion

We compared the run-time performance of the different languages. We ensured that each script used equivalent numbers of machines and other resources. Figure 9.4 shows total elapsed wall-clock time for each of the scripts. Jaql and SYRql required pre-processing of the data to convert the N-Triple RDF data into JSON-LD. The conversion, which took 40 minutes, is only needed once upon data loading and then the data can be used by all queries. In general, our SYRql implementation shows encouraging results. It is comparable to the times that Jaql and Pig Latin showed. However, Hive outperformed all the other four systems significantly. The superior performance of Hive was also reported in [151].

Both SYRql and Jaql can evaluate triple patterns that share the same subject together due to their underlying data model and their use of JSON-LD. Pig, on the other hand, evaluates each triple pattern individually and then joins the results. We believe that this is the main reason for the better performance that Jaql and SYRql generally achieved in comparison to Pig despite the maturity and the larger developers community that Pig enjoys.

Examining the generated MapReduce jobs, it was observed that Jaql and SYRql generated similar sequences of jobs. However, SYRql computes results for both graphs and bindings as specified in the underlying data model. This results in more computation to be done. Nevertheless, separating bindings and graphs optimisations helped speeding up many operators through reading and processing less data, particularly when rewriting rules can be applied to reduce the required computation.

We speculate that the superior performance of Hive is mostly due to its efficient join performance. Hive join optimisations such as conversion to map-only joins can be applicable when the joined relations are small in size. Additionally, for aggregation queries, Hive computes map-side partial aggregations using a combiner, an optimisation we plan to integrate in our future work.
In summary, SYRql implementation showed a good performance that will hopefully encourage users to try it. Moreover, SYRql scripts contained 50% less lines than Pig Latin scripts and 42% less than Jaql scripts. The result provided is a feasibility proof of SYRql performance rather than a rigorous performance comparative one. The various languages studied utilise different data models and different querying languages. Furthermore, these systems are currently under heavy development and continuously add performance improvements.

9.5 Related Work

A large number of declarative languages were introduced recently as part of the big data movement. These languages vary in their programming paradigm, and in their underlying data model. Pig Latin [125] is a dataflow language with a tabular data model that also supports nesting. Jaql [16] is a declarative scripting languages that blends in a number of constructs from functional programming languages and uses JSON for its data model. HiveQL [157] adopts a declarative syntax similar to SQL and its underlying data model is a set of tables. Other examples of languages include Impala\(^{11}\), Cascalog\(^{12}\), Meteor [77] and DryadLINQ [174]. [151] presented a performance as well as a language comparison of HiveQL, Pig Latin and Jaql. [142] also compared a number of big data languages but focuses on their compilation into a series of MapReduce jobs.

In the semantic web field, SPARQL is the W3C recommended querying language for RDF. A number of extensions to SPARQL were proposed in the literature to support search for semantic associations [10], and to add nested regular expressions [130] for instances. However, these extensions do not change the pure declarative nature of SPARQL. There are also a number of non-declarative languages that can be integrated in common programming languages to provide support for RDF data manipulation [126, 150]. In the more general context of graph processing languages, [170] provides a good survey.

We provided a more detailed presentation of big data languages and RDF query languages in Chapter 2.

\(^{11}\)https://github.com/cloudera/impala  
\(^{12}\)http://cascalog.org/
9.6 Summary

Based on RDF.co, this chapter presented SYRql, a dataflow language for large scale processing of RDF data. SYRql blended concepts from both SPARQL and big data languages. Similar to other big data processing languages, SYRql defines a small set of basic operators that are amenable to parallelisation. On the other hand, SYRql adopts a graph-based data model and supports pattern matching as in SPARQL. An implementation of SYRql on top of the MapReduce platform was described. Results of a performance comparison between SYRql implementation and other existing big data languages were also discussed in this chapter.
Chapter 10

Graph Aggregation in SPARQL

Of the operators defined in RDF.co, graph aggregation, dubbed Gagg, is the only operator that has no direct equivalent in SPARQL. In this chapter, we propose adding support for Gagg to SPARQL. We motivate the need for a graph aggregation operator in Section 10.1 and provide a formal definition of it based on existing SPARQL operators (Section 10.2). Section 10.3 describes an algorithm for efficient evaluation of Gagg expressions. Furthermore, we demonstrate the expressivity and efficiency of Gagg in our evaluation (Section 10.4).

10.1 Motivation

Graph aggregation condenses a large graph into a structurally similar but smaller graph by collapsing vertices and edges. Aggregating RDF graphs has been used to induce schemas [22, 93, 104], produce descriptive statistics [13, 21, 113] and build indices [92, 120]. Graph aggregation was also used to rank RDF datasets by aggregating their resources [49]. Moreover, graph aggregation was used to provide business intelligence on top of graph data [19, 33, 131, 177], in social network analysis [56], and in bibliometrics [58, 85].

Aggregation of RDF graphs can be achieved using existing SPARQL 1.1 [72] operators. However, aggregating a graph requires a complicated single SPARQL query (a combination of sub-queries, CONSTRUCT and GROUP BY operators) or a series of SPARQL queries to aggregate nodes and edges (as done in [113] for instance). Such queries become complicated and verbose and therefore hard
to write, debug, and optimise. Having graph aggregation as a first-class operator simplifies query writing and optimisation.

Figure 10.1 provides a motivating example for graph aggregation in SPARQL. It shows an example bibliographic RDF data describing some papers and their authors. One might be interested in studying the co-authorship structure between authors or at a more coarse-grained level, between organisations (Figure 10.3a). Similarly, one might choose to aggregate the data to study the structure of citations between authors, organisations, or conferences (Figure 10.3c). Such expressivity requires flexibility in defining the dimensions of aggregation and the relationships (e.g., co-authorship). This chapter describes how Gagg adds this expressivity to SPARQL by building on an existing core operator in SPARQL, namely basic graph patterns.

### 10.2 Model

Similar to graph aggregation in RDF.co, Gagg is defined in two-steps as shown in Figure 10.2. Firstly, nodes and edges in the original graph $G$ are grouped together in an intermediate graph that we call the *grouped graph*. Secondly, the grouped graph is reduced into an *aggregated graph*. 
Chapter 10. Gagg

The first step is based on a set of dimensions that defines how nodes and edges are grouped together. For example, in Figure 10.1 the resources :a1 and :a2 are grouped together when aggregating authors by the organisation they are member of. As we detail later, we use SPARQL operators as a flexible and powerful way to define grouping criteria.

The second step condenses the grouped graph into the final aggregated graph. This operation is done by what we call a graph reduce function. Gagg does not impose a specific structure on the final result RDF graph. Instead, a template similar to that of SPARQL construct queries is used by the graph reduce function to structure the final results. In this template, typical aggregate functions such as sum and count, can be applied to the nodes and edges in the grouped graph. It is worth pointing out that the grouped graph is not an RDF graph as its nodes represent sets of resources, while the aggregated graph is an RDF graph that can be used as input for further processing.

### 10.2.1 Preliminaries

Similar to the notation used across this thesis, we assume the existence of two disjoint countably infinite sets: \( U \) (URIs [15]), \( L \) (literals). \( T = U \cup L \) is the set of RDF terms (i.e., \( T \) is the set of all URIs and literals). We define an RDF triple\(^1\) as a triple \((s,p,o) \in U \times U \times T\). An RDF graph is a set of triples.

Moreover, we assume the existence of a set of variables \( V \) disjoint from the set of RDF terms \( T \), i.e., \( V \cap T = \emptyset \). A triple pattern \((s,p,o)\) is a triple where any of the three components can be either a variable or a term, i.e., the triple \((s,p,o) \in (T \cup V) \times (T \cup V) \times (T \cup V)\). A basic graph pattern (BGP) is a set of triple patterns.

\(^1\)We only consider ground RDF graphs and therefore we do not consider blank nodes.
Basic graph patterns in SPARQL represent conjunctive queries. From henceforward, we use the conjunctive query notation\(^2\) \( q(\bar{x}) := t_1, \ldots, t_n \) where \( t_1, \ldots, t_n \) are triple patterns and therefore \( \{ t_1, \ldots, t_n \} \) is a BGP.

We denote by \( \text{Var}(q) \) the set of variables occurring in the query \( q \). The query head variables \( \bar{x} \) are called distinguished variables, and are a subset of the variables occurring in \( t_1, \ldots, t_n \), i.e., we have \( \bar{x} \subseteq \text{Var}(q) \).

We denote with \( \llbracket q \rrbracket_G \) the set of query answers of \( q \) on \( G \). For the evaluation of a query \( q \) against a graph \( G \), we refer the reader to the W3C Recommendation [72].

**Definition 10.1** (Join Query). Let \( q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n) \) be basic graph patterns which non-distinguished variables are pairwise disjoints. We call the query \( q(\bar{x}) := q_1(\bar{x}_1) \land \cdots \land q_n(\bar{x}_n) \) a join query of \( q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n) \), where each variable in \( \bar{x} \) appears in at least one of \( \bar{x}_1, \ldots, \bar{x}_n \).

A join query combines multiple BGPs and joins them based on their shared distinguished variables.

### 10.2.2 Graph Aggregation Operator

We build on the definitions introduced before to define a graph aggregation operator. Notice also that we adopt the OLAP terminology where items are grouped based on a set of dimensions and then measures associated with these items are aggregated per group. In Gagg, dimensions used for aggregation, measures that are to be aggregated, and the relations between nodes in the graph are all expressed as queries. In particular, a dimension is a query with two distinguished variables \( q(x,v) \) that defines how resources (bound to \( x \)) are grouped based on associated values (bound to \( v \)). Similarly, a measure is also a query with two distinguished variables \( q(x,m) \) that associates with each resource (bound to \( x \)) a value to be aggregated (bound to \( m \)). In Figure 10.3b, \( q(x,o) := (x,:\text{member},o) \) is one of the dimensions used to group authors while \( q(x,p) := (p,:\text{creator},x) \) is the measure associated with each resource as we are counting the papers written by each author.

A relation query is a query with four distinguished variables \( q(x,p,y,m) \) that defines related resources (bound to \( x \) and \( y \), respectively), labels the relation (via

\(^2\)We reuse here some of the notations and definitions from [41].
(a) Aggregation of people by the organisation, counting the number of papers authored by the organisation. The edges :co-authorship link organisations by the papers they authored.

(b) Aggregation of people by their organisation and position, counting the number of their papers. The edge :citation represents an author citing another via a paper.

(c) Aggregation of people by their organisation and papers by their conference. We count the number of authors in an organisation, and the number of papers in a conference. The edge :citation represents a link from an organisation to a conference, where an author cited a paper from a particular conference.

Figure 10.3: Summary graphs of the data in Figure 10.1. The number within parentheses report a count statistics of the aggregated data.
the value bound to \( p \), and determines the measure associated with this relation (bound to \( m \)). In Figure 10.3b, the relation query used is:

\[
q(\text{author1}, \text{:cite}, \text{author2}, \text{paper1}) := (\text{paper1}, \text{:creator}, \text{author1}) \land \\
(\text{paper2}, \text{:creator}, \text{author2}) \land \\
(\text{paper1}, \text{:references}, \text{refs}) \land \\
(\text{refs}, \text{:ref}, \text{paper2})
\]

Notice that in the previous query, \( \text{paper1} \) is used as a measure since we are interested in counting the papers. Additionally, the relation is bound to the constant value \( \text{:cite} \) to give it a readable name\(^3\).

**Definition 10.2 (Grouped Graph).** A grouped graph is a tuple \((V, A, l_V, \text{dimensions}, \text{measures})\) where \( V \) is a set of nodes, \( A \in V \times T \times V \) is the set of labelled edges, and \( l_V : V \rightarrow T \) is a node labelling function. \( \text{dimensions} : V \rightarrow 2^T \) and \( \text{measures} : V \cup A \rightarrow 2^T \) are two associated functions.

Figure 10.4 shows an example grouped graph with its associated \( \text{dimensions} \) and \( \text{measures} \) functions. Definition 10.5 defines how a grouped graph is related to a given RDF graph when it is aggregated.

**Definition 10.3 (Graph Reduce Function).** A graph reduce function \( f \) is a function that maps a grouped graph into an RDF graph.

We are now ready to provide a definition of the Graph Aggregation Query.

**Definition 10.4 (Graph Aggregation Query).** A graph aggregation query is a tuple \( Q = (D, M, E, N, R, f) \) such that:

- \( R \) is a relation query with distinguished variables \( x, p, y, o \);
- \( D \) is a set of dimensions such that for each dimension \( d \in D \) the distinguished variables of \( d \) are \( x, v \) for some unique variable \( v \);
- \( E \) is a set of dimensions such that for each dimension \( e \in E \) the distinguished variables of \( e \) are \( y, w \) for some unique variable \( w \);

\(^3\)For the sake of simplicity, we slightly violated conjunctive query notation by using a constant in the header.
Chapter 10. Gagg

Figure 10.4: The grouped graph that corresponds to the aggregated graph in Figure 10.3b. This graph aggregates authors in Figure 10.1 by their organisation and position

- $M$ and $N$ are two measure queries with distinguished variables $x, m$ and $y, n$, respectively;
- $f$ is a graph reduce function.

In the definition above, $R$ is meant to define related nodes in the typical form of (subject, predicate, object) with an associated measure. $D$ and $E$ are the sets of dimensions to group the subjects and objects defined by $R$. While the dimensions for subjects and objects can be the same, Figure 10.3c depicts an example where it is beneficial to group subjects and objects by different dimensions. Finally, $M$ and $N$ are the measures for subject and object nodes.

**Definition 10.5.** A grouped graph $G = (V, A, l_V, dimensions, measures)$ is the result of grouping an RDF graph $G$ according to the graph aggregation query $Q = (D, M, E, N, R, f)$ if the following holds:
1. \( \forall w \in \mathcal{V} \land a \in \mathcal{T} \) it is the case that \( a \in \text{dimensions}(w) \) if and only if \( \exists (r,a) \in [D]_G \) or \( (r,a) \in [E]_G \). In other words, in the grouped graph \( \mathcal{G} \) a value \( a \) will be in the dimensions of a node \( w \) if and only if \( a \) is associated with some node in the original RDF graph \( G \) according to a dimension in \( D \) or in \( E \). In this case we say that the node \( r \) in the original graph maps to the node \( w \) in the grouped graph and denote this as \( r \mapsto w \);

2. \( \forall u \in \mathcal{V} \land a \in \mathcal{T} \) it is the case that \( a \in \text{measures}(u) \) if and only if \( \exists (r,a) \in [M]_G \) or \( (r,a) \in [N]_G \) and \( r \mapsto u \). In other words, in the grouped graph \( \mathcal{G} \) a value \( a \) will be in the measures of a node \( u \) if and only if one of the measures \( M \) or \( N \) associates \( a \) to some node \( r \) in the original RDF graph \( G \) and \( r \) maps to \( u \);

3. \( \forall (u,p,v) \in \mathcal{A} \land a \in \mathcal{T} \) it is the case that \( a \in \text{measures}((u,p,v)) \) if and only if \( \exists (s,p,o,a) \in [R]_G \) such that \( s \mapsto u \) and \( o \mapsto v \). In other words, in the grouped graph \( \mathcal{G} \) a value \( a \) will be in the measures of an edge \( (u,p,v) \) if and only if \( R \) associates two nodes \( s \) and \( o \) in the original graph \( G \), such that the measure value of the association according to \( R \) is \( a \) and \( s \) and \( o \) map to \( u \) and \( v \), respectively.

### 10.3 Answering Graph Aggregation Queries

The evaluation of a graph aggregation query is performed in three steps: (1) building a binding table that is the solution of the queries defined by the dimensions, relation and measures; (2) building a grouped graph from the binding table; and (3) applying the reduce function over the grouped graph to achieve the final results.

#### 10.3.1 Binding Table

Evaluating a query in SPARQL results in a binding table as detailed in [72]. The first step to answer a graph aggregation query \( Q = (D, M, E, N, R, f) \) is to combine and evaluate the query:

\[ [q]_G \] is the solution of query \( q \) against graph \( G \). We extend the notion here to a set of queries where the result is the join query as defined before.
The result is a binding table $\mathcal{B}$ which is a set of rows. If $r$ is a row in $\mathcal{B}$ and $x$ is a variable, we use $r[x]$ to refer to the value bound with the variable $x$ in $r$. The evaluation of this SPARQL query can be delegated to an existing SPARQL engine. This delegation exploits the best practices and optimisation techniques already built in existing SPARQL engines.

### 10.3.2 Building the Grouped Graph

Algorithm 2 describes how starting from the binding table obtained from the previous step, a grouped graph compatible with the Definition 10.5 can be built. The function $\text{GetGroupNode}(v)$ creates a node corresponding to the dimensions values $v$, updates the dimensions mapping of $\mathcal{G}$ and adds the created node to $\mathcal{V}$. If a node corresponding to the dimension values already exists in $\mathcal{V}$, the function just returns the node. Notice that the $\text{GetGroupNode}$ function implements the maps to (↦) relationship from nodes in original graph to those in the grouped graph as specified in the first condition of Definition 10.5. Furthermore, a value is added to the set of measures of a node or an edge only if it is matched by the corresponding query and the maps to relationship (lines 6-8). This ensures the other two conditions of Definition 10.5.

The $\text{GetGroupNode}$ function needs some hashing-based structure that maps dimension values to nodes and keeps the measures associated with each node and edge. Therefore, the complexity of $\text{GetGroupNode}$ is $O(1)$. Consequently, Algorithm 2 has a complexity of $O(|\mathcal{B}|)$ as it scans the binding table only once.

### 10.3.3 Applying the Graph Reduce Function

The graph reduce function applies an aggregate function on the set of measures and restructure the grouped graph into a result RDF graph. Structuring the final
Algorithm 2: Aggregated graph creation from the table of bindings

**Data:** The binding table $\mathcal{B}$ and the graph aggregation query $(D, M(x, m), E, N(x, n), R(x, p, y, o), f)$

**Result:** The grouped graph $\mathcal{G} = (\mathcal{V}, \mathcal{A}, l, \mathcal{V}, \text{dimensions}, \text{measures})$.

for $r \in \mathcal{B}$ do // For each row in the binding table
  // Retrieve grouped nodes corresponding to the dimensions
  $u \leftarrow \text{GetGroupNode}(r[D])$
  $v \leftarrow \text{GetGroupNode}(r[E])$
  // create an edge
  $e \leftarrow (u, r[p], v)$
  $\mathcal{A} \cup e$

  // Add measures
  $\text{measures}(u) \leftarrow r[m]$
  $\text{measures}(v) \leftarrow r[n]$
  $\text{measures}(e) \leftarrow r[o]$
end

result is not specified by Gagg but can be defined using a template similar to the one used in CONSTRUCT queries in SPARQL. For example, a user might choose to represent relations along with their associated measures in the aggregated graph using the VoID ontology\(^5\) or using RDF Reification. Using a template, both of these options can be achieved. SPARQL aggregate functions can be used in the template to aggregate measures in the grouped graph.

10.4 Evaluation

In this section, we discuss the expressivity of Gagg operator and present an experimental performance comparison with plain SPARQL queries.

10.4.1 Expressivity

We have used Gagg to express type summary as defined in [22] and to reproduce VoID statistics similar to the results provided by previous systems [13, 113]. Furthermore, Gagg was applied to bibliographic data to compute a number of bibliometrics as shown in the next section. Finally, we have used Gagg to aggregate

\(^5\)https://www.w3.org/TR/void/
the LOD Cloud\textsuperscript{6}. As this data is available in RDF\textsuperscript{7}, we were able to generate different versions of the LOD diagram by aggregating datasets by their topics, licenses, publishers, etc. Gagg is expressive enough to aggregate datasets of the LOD cloud and count them or sum their triples counts\textsuperscript{8}. Similarly, it can count the number of linksets and sum the number of interlinking triples. This provided views over the LOD cloud from a variety of perspectives.

### 10.4.2 Performance Evaluation

We extended the query algebra in Apache Jena\textsuperscript{9} with the Gagg operator. The implementation builds the intermediate graph and then apply aggregate functions whose implementations are re-used from Jena. We compare the performance of the usage of Gagg to that of standard SPARQL. In particular, we report the average running time of the four following approaches:

- **reduced** provides incomplete results as it aggregates only the relationships and not the subjects and objects. This is included as a baseline to quantify the extra time needed by the other approaches to get the full results.
- **fullSparql** builds the aggregated graph using one CONSTRUCT SPARQL query. This query contains three sub-queries such that the first one defines the relationship, the second aggregates and counts subjects and the final one aggregates and counts objects.
- **3Sparqls** uses three separate CONSTRUCT SPARQL queries to build the aggregated graph. Similar to the fullSPARQL approach, one query defines the relationship and the other two queries aggregate subjects and objects. However, on contrary to the three nested queries in fullSPARQL, the three queries in 3Sparqls are evaluated separately. Notice that these queries need to rely on some characteristics of the data or some hashing function to assure that the results of the three queries use the same identifiers for the aggregated nodes. Therefore, writing these queries was relatively hard. The final result is the union of the three graphs resulted from the queries. However, to

\textsuperscript{6}http://lod-cloud.net/
\textsuperscript{7}http://lod-cloud.net/data/void.ttl
\textsuperscript{8}The scripts to generate the aggregated graphs of the LOD Cloud diagram are available at https://github.com/fadmaa/rdf-graph-aggregation
\textsuperscript{9}https://jena.apache.org/ version 2.12.0.
prevent introducing extra penalty on the running time, the values we report
do not include the time needed to union the three results.

**Gagg** uses the **Gagg** operator we added to Jena.

We used JUnitBenchmarks\(^{10}\) to run the evaluation. JUnitBenchmarks performs
JVM warm-up phases and repeat the execution multiple times to enhance the
reliability of the reported times. The evaluation was run on a 4 core machine
running Linux (3.18.1-3) at 2.60GHz with 8Gb of RAM. The JAVA version is
1.7.0_71.

We experimented with BSBM [18] and SP\(^2\)B [145] datasets, varying the sizes of
data. Two sets of queries are used to perform two tasks: building a type summary
and calculating some bibliometrics-based summary.

### 10.4.2.1 Type Summary

For this graph aggregation query, all resources in the RDF data are grouped by
their types (i.e., the values of `rdf:type`) and all relations between resources are
grouped. The summary reports the number of instances per type and the number
of relations of each type that exists between instances of two classes. This type of
statistics are the ones reported in VoID statistics and in the RDF graph summary
in [22].

In the corresponding **Gagg** queries, the set of dimensions of subjects and objects
is (`?x,rdf:type,?t`) and the relationship query is (`?x,?p,?y`).

### 10.4.2.2 Bibliometrics

We report on three graph aggregation queries that were evaluated on top of SP\(^2\)B
data:

**co-authorship** gets the graph structure of co-authorship. Nodes in the summary
graph represent authors along with the number of papers they published
while edges represent co-authorship between them along with the counts.

\(^{10}\)http://labs.carrotsearch.com/junit-benchmarks.html
citation gets the graph structure of author citation. Nodes in the summary graph represent authors along with the number of papers they authored while edges represent citation across authors along with their counts.

conf-citation gets the graph structure of citation among papers grouped by the conference they are published in. Nodes in the summary graph represent conferences along with the number of papers published in each while edges represent citation across conferences along with their counts.

These queries are reported in Appendix B.

10.4.3 Discussion

The average running times to compute the Type Summary queries are reported in Table 10.1, while the average running times of the bibliometrics queries are shown in Table 10.2. Entries marked with N/A failed to finish as the machine ran out of memory or the execution took too long (more than 5 minutes).

In general fullSparql approach showed the worst performance among the tested approaches. This is not surprising giving that dimensions and measures are evaluated three times in the fullSparql because results need to be aggregated along different dimensions each time, i.e., for subject, object and relation. Results of the three sub-queries of fullSparql also need to be joined together. In particular, the dramatic growth in the TypeSummary queries is due to the need for joining the results of the three sub-queries before aggregation. Because we are interested in aggregating all relations in the graph to compute a type summary, the sub-queries are not selective and produce large intermediary results. This is an extra overhead cost that both Gagg and 3Sparqls approaches avoid. In comparison to fullSparql, Gagg achieved up to two orders of magnitude improvement in response time.

The performance improvement of Gagg of about a 2.5 factor in comparison to 3Sparqls is reasonable. Gagg builds and scans the binding table once instead of three times. Nonetheless, it still performs the three aggregations.

Finally, in comparison to the reduced approach, the overhead that Gagg adds to achieve the full aggregated graphs is small. We remind that reduced approach
### Table 10.1: Average running times of type summary queries

<table>
<thead>
<tr>
<th>dataset-size (# triples)</th>
<th>fullSparql (s)</th>
<th>3Sparqls (s)</th>
<th>reduced (s)</th>
<th>Gagg (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsbm-5K</td>
<td>0.08</td>
<td>0.06</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>bsbm-190K</td>
<td>9.84</td>
<td>1.25</td>
<td>0.42</td>
<td>0.55</td>
</tr>
<tr>
<td>bsbm-370K</td>
<td>31.88</td>
<td>2.82</td>
<td>1.00</td>
<td>1.13</td>
</tr>
<tr>
<td>bsbm-1.8M</td>
<td>454.07</td>
<td>13.48</td>
<td>4.37</td>
<td>5.61</td>
</tr>
</tbody>
</table>

(a) BSBM

<table>
<thead>
<tr>
<th>dataset-size (# triples)</th>
<th>fullSparql (s)</th>
<th>3Sparqls (s)</th>
<th>reduced (s)</th>
<th>Gagg (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp2b-50K</td>
<td>0.42</td>
<td>0.22</td>
<td>0.08</td>
<td>0.09</td>
</tr>
<tr>
<td>sp2b-100K</td>
<td>0.84</td>
<td>0.44</td>
<td>0.15</td>
<td>0.16</td>
</tr>
<tr>
<td>sp2b-500K</td>
<td>4.75</td>
<td>2.19</td>
<td>0.75</td>
<td>0.84</td>
</tr>
<tr>
<td>sp2b-1M</td>
<td>10.32</td>
<td>4.84</td>
<td>1.56</td>
<td>1.8</td>
</tr>
</tbody>
</table>

(b) SP²B

<table>
<thead>
<tr>
<th>dataset-size (# triples)</th>
<th>fullSparql (s)</th>
<th>3Sparqls (s)</th>
<th>reduced (s)</th>
<th>Gagg (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp2b-50K</td>
<td>3.93</td>
<td>0.18</td>
<td>0.053</td>
<td>0.09</td>
</tr>
<tr>
<td>sp2b-100K</td>
<td>4.05</td>
<td>0.20</td>
<td>0.05</td>
<td>0.08</td>
</tr>
<tr>
<td>sp2b-500K</td>
<td>4.02</td>
<td>0.18</td>
<td>0.05</td>
<td>0.08</td>
</tr>
<tr>
<td>sp2b-1M</td>
<td>N/A</td>
<td>6.80</td>
<td>1.04</td>
<td>2.62</td>
</tr>
</tbody>
</table>

(a) Co-authorship

<table>
<thead>
<tr>
<th>dataset-size (# triples)</th>
<th>fullSparql (s)</th>
<th>3Sparqls (s)</th>
<th>reduced (s)</th>
<th>Gagg (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp2b-50K</td>
<td>1.15</td>
<td>1.02</td>
<td>0.32</td>
<td>0.5</td>
</tr>
<tr>
<td>sp2b-100K</td>
<td>4.89</td>
<td>4.29</td>
<td>1.44</td>
<td>1.62</td>
</tr>
<tr>
<td>sp2b-500K</td>
<td>94.06</td>
<td>86.39</td>
<td>29.41</td>
<td>30.7</td>
</tr>
</tbody>
</table>

(b) Citation

<table>
<thead>
<tr>
<th>dataset-size (# triples)</th>
<th>fullSparql (s)</th>
<th>3Sparqls (s)</th>
<th>reduced (s)</th>
<th>Gagg (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp2b-50K</td>
<td>0.14</td>
<td>0.17</td>
<td>0.05</td>
<td>0.06</td>
</tr>
<tr>
<td>sp2b-100K</td>
<td>0.67</td>
<td>0.68</td>
<td>0.22</td>
<td>0.23</td>
</tr>
<tr>
<td>sp2b-500K</td>
<td>N/A</td>
<td>29.31</td>
<td>9.7814</td>
<td>9.8091</td>
</tr>
</tbody>
</table>

(c) Conf-citation

Table 10.2: Average running times of bibliometrics queries on the SP²B data

gives incomplete results, since it does not aggregate the subject and object resources.
10.5 Related Work

On-Line Analytical Processing [38] (OLAP) has been first proposed as a way for people to analyse multi-dimensional data. Several works aim at analysing graph data using the OLAP paradigm. The Graph Cube [177] paper considers only simple graphs, while we target more complex graphs that are possible with the RDF graph model. The work in [33] proposes the aggregation of graphs which changes the topology of the graph. However, we allow a more flexible definition of the aggregation dimensions in our approach. In this work, we introduce the graph aggregation as a graph operator that is anchored into SPARQL, allowing the graph aggregation to be part of a wider graph analysis flow.

The challenge of graph analysis has been studied within the Semantic Web community as well. The RDF Analytics [41] paper introduces an analytical schema over the data. This schema allows writing analytical queries over the data, abstracting from the actual structure of the data. However, unlike Gagg, this approach does not generate a graph as the analytical query output, thus preventing the use of the results as the input of other graph operations.

The process of graph summarisation which represents a graph with a smaller graph that is homomorphic to the original graph is investigated in [22, 159, 176]. Such works write their own custom code to achieve graph aggregation and can, therefore, benefit from adding Gagg to the family of SPARQL operators.

10.6 Summary

In this chapter, we introduced a graph aggregation operator, Gagg, to SPARQL. We defined the operator as a two-steps process, where the graph is first grouped based on some dimensions, and then reduced into an aggregated graph. The aggregated graph exhibits groups of dimensions and relations between such groups, as well as statistics associated to the groups and links. Our definition integrates with existing SPARQL operators, making it possible to compose Gagg with other SPARQL operators. The actual aggregation of the graph can be fully customised by a user via the use of graph pattern queries. The structure of the results can also be controlled by the user via the use of a template similar to that of SPARQL
construct queries. We showed in our evaluation that Gagg improves significantly over the performance of plain-SPARQL graph aggregation.
Chapter 11

Summary & Conclusions

In the big data world, RDF data has an important role to play. Its semantic richness and flexible data model can facilitate integrating data coming from various sources and in different formats. Consequently, the number of big RDF datasets is growing. Examples include Freebase\(^1\), DBpedia\(^2\), and Wikidata\(^3\). Managing and querying this data is a challenge.

The challenge of managing and querying RDF data attracted significant attention since the early days of the Semantic Web (e.g., [20, 89]). In 2008, SPARQL 1.0 became an official W3C Recommendation and a newer specification, SPARQL 1.1, was published in 2013. SPARQL enjoys a wide adoption in the Semantic Web community and various implementations of SPARQL engines are available (e.g., AllegroGraph\(^4\), OpenLink Virtuoso\(^5\), and Ontotext\(^6\)).

In the broader data management field, the growth of both the size and the formats of the available data posed new challenges to data querying languages. This resulted in a number of new data management systems and querying languages typically classified as NoSQL systems. Particularly for big data, various languages were developed. Examples of big data languages include PIG Latin from Yahoo [125], DryadLINQ from Microsoft [174], Jaql from IBM [16], HiveQL from

\(^1\)https://developers.google.com/freebase
\(^2\)http://dbpedia.org
\(^3\)http://www.wikidata.org
\(^4\)http://franz.com/agraph/allegrograph
\(^5\)http://virtuoso.openlinksw.com
\(^6\)http://ontotext.com
Facebook [157], FlumeJava [26] from Google, and Apache Spark [1]. These languages were used to query large volumes of RDF data [66, 143, 156].

Therefore, when analysing large RDF datasets, users are left mainly with two options: using SPARQL, the main query language for RDF, or using an existing non-RDF-specific big data language. This thesis argued that each of these two approaches has its own limitations.

In this thesis we set out to explore defining a dataflow language for processing large amounts of RDF data. Dataflow programs represent a set of transformation of data where results of one transformation form input of the next one. While these transformations are still declarative, the flow is clear and defined by the user program. Dataflow languages are inherently parallel and can work well in decentralised systems [45, 86].

In this thesis, the language was used as a vehicle to investigate a new data model for RDF processing and a distributed physical layout for RDF graphs. The three main areas this thesis focused on are data models, data query languages, and graph partitioning. Corresponding contributions are shown in Figure 11.1 in the context of a layered big data architecture. All these contributions are oblivious to the specific distributed computing engine used and can be implemented on different shared-nothing computing engines such as MapReduce, Apache Spark, or Apache Flink.\footnote{https://flink.apache.org}

We next summarise our research questions and hypotheses in each of these three areas.

**On Data Models**

When defining a language, the underlying data model plays a critical role in ensuring the clarity and preciseness of the language semantics and in enabling reasoning about constructs of the language. A data mode provides the notation to describe data and a set of operations used to manipulate that data.

For our dataflow language, the widely used and highly cited SPARQL algebra model was the first candidate data model. However, the SPARQL algebra is not fully composable. The current SPARQL algebra transitions, via triple pattern
matching, from graphs (i.e., the initial inputs) to sets of bindings (which are basically tables resulting from pattern matching). Subsequently, further operators such as Join, Filter, and Union are applied on sets of bindings. In other words, the flow is partly hard-coded in the SPARQL algebra. In a dataflow language, the dataflow is guided by the user and cannot be limited to the way the SPARQL algebra imposes. Our approach was to keep both graph and binding for each expression. We set out to define such a data model, dubbed RDF.co, and study its algebraic properties.

**Hypothesis.** Pairing bindings and graphs in expression values allows defining a fully composable data model for querying RDF data. The algebraic properties of this model provide optimisation opportunities.

**On Graph Partitioning**

In response to the increasing heterogeneity of graphs we wanted to explore partitioning methods that take into account the structure of the graph and the anticipated work load. Partitioning algorithms used in distributed graph systems, such as ParMETIS [90] and balanced vertex-cut [63] algorithms, do not match the heterogeneity of recent graphs. We proposed using pattern matching to partition...
the graph in a way that matches the anticipated query load and set out to research
the possibility of realising this and study the implication on performance.

**Hypothesis.** When prior knowledge is available about query load, partitioning
an RDF graph using pattern matching enhances performance of query answering.

## On Data Query Languages

To enable users to express their dataflow programs using RDF\_co, a language that
provides syntactical constructs is essential. Such language needs to allow defining
individual RDF\_co operators and combining them together. Furthermore, dataflow
programs should be compiled into an executable form and run on top of a shared-
nothing distributed platform.

In comparison to general purpose big data languages, a dataflow language designed
specifically for RDF helps generating scripts that are easier to understand and
debug. Therefore, we want to ensure that an implementation of such language
also achieves similar level of performance to match existing big data languages.

**Hypothesis.** A dataflow language for Big RDF Data allows declarative ma-
nipulation of RDF data and provides a similar performance to existing big data
languages when implemented on top of the MapReduce platform.

## 11.1 Findings

### 11.1.1 On Data Models

We introduced RDF\_co, a data model that defines a pair of binding and a graph
in the value of each expression. RDF\_co, similar to other big data models, defines
a small set of basic operators that are amenable to parallelisation. Each of these
operators transforms a set of binding-graph pairs into a new set of binding-graph
pairs and, therefore, ensures full compositability. The set of operators defined in
RDF\_co are: bAccess, PROJ, bExt, gExt, MATCH, FILTER, and CROSS. These oper-
ators allow querying and manipulating RDF data. We provided a formal defini-
tion of the syntax and semantics of these operators. We also proved that each of
RDF\_co operators cannot be defined using the other operators (the set of operators
is minimal). Additionally, **RDF.co** defines two aggregation operators: **AGG** provides aggregation capabilities similar to that of SPARQL and **Gagg** allows condensing graphs into smaller aggregated graphs.

In discussing the relationship of **RDF.co** to SPARQL, we showed that **RDF.co** operators can define SPARQL datasets, named graphs, evaluating a pattern against a particular named graph, and expressing SPARQL 1.1 SELECT and CONSTRUCT queries with basic graph patterns, filters, and assignments. Furthermore, using **RDF.co** one can also express some SPARQL nested queries. In particular, **RDF.co** can express SPARQL queries in which a SELECT query is used in the place of a graph pattern and SPARQL queries that use a CONSTRUCT query in the FROM clause (not part of the standard SPARQL).

We studied the unique algebraic properties of **RDF.co**. These properties, when interpreted as rewriting rules, provide theoretical foundation needed to apply cost-based query optimisation. We focused on two sets of properties. Firstly, properties that are results of the ability to cascaded triple patterns matching in **RDF.co**. We presented three main properties: triple pattern elimination, triple pattern insertion, and triple pattern push-down. Secondly, we studied the properties that are results of the use of two-component values (graph and binding components) in **RDF.co**. In this regards, we defined five characteristics an operator can have. These characteristics are b-idempotent, g-idempotent, component-separable, b-independent, and g-independent. Table 11.1 summarises the characteristics each of the **RDF.co** operators has. These algebraic properties allow rewriting many **RDF.co** expressions into simpler ones that can be evaluated more efficiently.

<table>
<thead>
<tr>
<th></th>
<th>b-idempotent</th>
<th>g-idempotent</th>
<th>component-separable</th>
<th>b-independent</th>
<th>g-independent</th>
</tr>
</thead>
<tbody>
<tr>
<td>bAccess</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PROJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>bExt</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gExt</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MATCH</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FILTER</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CROSS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 11.1**: Summary of **RDF.co** operators characteristics
11.1.2 On Graph Partitioning

We presented pattern-based graph partitioning. This approach allows utilising available knowledge about the data and the query load to guide graph partitioning. In comparison to the widely used min-cut partitioning method, pattern-based partitioning accommodates graph structure and edge labels. In comparison to recent n-hops partitioning methods of RDF data [81, 99], pattern-based partitioning allows tailoring the partitioning and replication to anticipated query loads instead of exhaustive replication used by those previous methods.

To utilise the data locality provided by pattern-based partitioning when answering queries, we reduced the problem of query decomposition to that of view-based query answering. We defined graph-centric criteria sufficient for deciding on query answerability against a pattern-based distributed graph.

Our experimental results with a MapReduce-based implementation showed that pattern-based partitioning provides significant improvement over random and star-shaped partitioning. It also shows that tailored patterns can achieve similar performance to n-hop methods with much less replication.

11.1.3 On Data Query Languages

We defined SYRql, a dataflow language on top of RDF.co. We provided a full implementation of SYRql on top of MapReduce. Our implementation includes all components required to allow the use of SYRql by end users: parsing, producing a logical plan, optimising the logical plan, producing a physical plan, and finally execution as a series of MapReduce jobs. Our implementation makes use of RDF.co algebraic properties and the underlying pattern-based partitioning. Experimental results compared SYRql performance to that of existing big data languages, namely HiveQL, Pig Latin, and Jaql. Different than these languages, SYRql has to maintain and process more data due to the pairing of graphs and bindings that are essential to ensure full composability. Nevertheless, utilising RDF.co properties, SYRql showed similar performance to that of Pig Latin and Jaql.

Finally, the set of operators that we defined in RDF.co have equivalent operators in the SPARQL algebra. The only operator that has no direct equivalent in SPARQL is \texttt{Gagg}, the graph aggregation operator. We showed how this operator
can be integrated into the SPARQL algebra and how such integration provides both expressivity and performance advantages.

11.2 Limitations

11.2.1 On Data Models

*Missing operators.* As stated before, RDF.co cannot express SPARQL OPTIONAL matching and set operators, particularly UNION. OPTIONAL clauses in SPARQL is the main differentiation between SPARQL and traditional relational query languages. An interesting future direction would be to add optional matching to RDF.co and study the effect of this addition on the model expressivity, complexity, and algebraic properties.

*Bag semantics.* RDF.co was defined only in set semantics (i.e., no duplicates allowed in values). While set semantics is typically used for formal definition of data models, in practice bag semantics are more suitable. Bag semantics do not require the costly step of removing duplicates. We did not provide a bag semantics definition of RDF.co and did not state whether the characteristics and algebraic properties reported for set semantics still hold for bag semantics.

11.2.2 On Graph Partitioning

*High cost of partitioning using complicated patterns.* As shown in the partitioning algorithm, a full scan of the data is required for extra triple in the partitioning pattern. This means that using complicated and more expressive patterns results in longer loading time. While this is a onetime loading cost, it still can be limiting in scenarios where fast loading of data is necessary.

*Selecting partitioning patterns.* Beyond utilising existing knowledge about the data and the anticipated query load, we did not provide a systematic method for selecting suitable patterns that can be used for partitioning.
Only extended star-shaped patterns are supported. While the definition of pattern-based partitioning can accommodate any pattern, we only showed algorithms and implementation for realising star-shaped patterns that are enriched with further replicated triples.

11.2.3 On Data Query Languages

Usability evaluation of SYRql. We evaluated the performance of SYRql programs as compared to other existing big data languages but did not evaluate the usability of the language.

Further optimisation of SYRql. As stated before, we did not implement the optimisations related to cascading triple patterns in SYRql.

11.3 Directions for Future Research

11.3.1 On Data Models

Algebraic properties of basic graph patterns. The algebraic properties we studied focused on expressions containing a single triple pattern matching. It would be interesting to study how these properties can be generalised to the more complex basic graph patterns. It is worth pointing out here that our work on pattern-based graph partitioning can be a good starting point as it effectively studied a relationship between basic graph patterns.

Implementing SPARQL using RDF.co. We studied the expressivity of RDF.co as compared to SPARQL. Using the RDF.co model to implement a SPARQL engine can provide the opportunity to apply RDF.co rewriting rules and to utilise the full composability in caching and reusing previous queries results. It would be interesting to study if such implementation can improve the efficiency of SPARQL engines.
11.3.2 On Graph Partitioning

In addition to the challenges of integrating pattern matching partitioning within existing systems and experimentally evaluating them, we believe that there exists a number of further directions that are worth investigating:

**Fast partitioning.** Pattern-based partitioning is expensive due to the need of multiple passes over the data and the need to join possibly large intermediate results. One direction that can be investigated to speed up partitioning is the work on best-effort and approximate graph pattern matching (e.g., [160, 162]). The ease of integrating approximate and incomplete results however varies based on the underlying platform. It is straightforward to integrate in systems that abstract communication between tasks from the user such as GraphX and GraphLab. On the other hand, it is more challenging to guarantee correctness using incomplete or approximate partitions in MapReduce or BSP systems where communication is explicitly controlled by the user.

**Adaptive partitioning.** Repartitioning can be performed as a side-product of query processing. The fact that partitioning is defined using the same constructs of queries (i.e., patterns) can make such adaptive partitioning more straightforward. Work on adaptive indexing [70, 82] (a.k.a., database cracking) can be utilised.

**Incremental partitioning.** It is very expensive to require a complete repartitioning every time the data is updated and therefore performing partitioning in a stream fashion can be necessary. There exists some work on incremental answering of graph pattern matching [53] and on graph stream processing [98] that can be leveraged.

11.3.3 On Data Query Languages

**Usability evaluation of SYRql.** We evaluated the performance of SYRql programs as compared to other existing big data languages. Further evaluation of the usability of SYRql would be interesting. SYRql can be evaluated using a questionnaire (as done in [36]) or by direct comparison to SPARQL (as done in [28] that compared Java and C#).

**Implementing SYRql on Apache Spark.** Our current implementation of SYRql is based on MapReduce. Apache Spark provides an in-memory computation
model that is richer than that of MapReduce. Implementing SYRql on top of Apache Spark can provide significant performance improvements.

11.4 Afterword & Lessons Learnt

The Elements of the Big Data:

Big data has been frequently characterised by the three Vs: Volume, Velocity, and Variety. While we believe that these are important characteristics of the big data revolution, we argue that the data is only a part of the whole story. The increasing availability of the data resulted in a higher expectation and a stronger desire to utilise this data beyond simple analytics and reporting. This need is reflected in the increasing emphasis on advanced analytics using machine learning and data mining. Furthermore, this was accompanied by the need to bring wider audience, beyond highly-skilled professionals, into effective use of the big data. To summarise, we believe that the big data revolution is the combined result of three elements: (i) there is more data, (ii) we want more from data, and (iii) there is more diverse users of the data.

In the light of these three elements, the popularity of MapReduce systems can be understood as a general purpose system that provides a simple API and is capable of handling massive amounts of data.

Big Data Value Proposition:

It has been argued that big data could enable a new method of scientific discovery. Others also argued that big data can revolutionise data analysis by dropping the need for random sampling and allowing analysis of whole populations. On the contrary, big data systems were frequently criticised in the database community. For instance, MapReduce was described as a major step backward and a number of critical evaluations of big data systems appeared in database literature (for example, [128]).

[9] See Chapter 2 in [117]
Nevertheless, we argue that the real value proposition of big data is in enabling unprecedented scalability as a cornerstone for use cases that were not possible before. It is important to keep this value proposition when comparing big data systems with other systems that have some built-in scalability limits.

**Big Data and Databases:**

It is interesting to notice that big data systems and databases are increasingly borrowing from each other. For instance, many databases now support HDFS as a data source.\(^\text{11}\) On the other hand, many big data systems support typical database features such as structured data, transactions, and indices.\(^\text{12}\) This can be seen as a result of maturity of big data systems and a result of a healthy cross-pollination. On the other hand, this can be viewed as a result of a pressure to monetise big data systems that shifted their focus to the multibillion business intelligence markets. This, in turn, risks losing an opportunity to advance unstructured data processing and to develop a more generic data processing model.

**Evaluating Big Data Systems:**

Evaluating big data systems poses unique challenges. These challenges include handling a larger data size, the distributed infrastructure, and the large number, and effect, of available configuration parameters. Furthermore, evaluating big data systems needs to pay particular attention to balancing the typical two competing metrics: scalability and performance. While it is important to keep in mind the value proposition of enabling unprecedented scalability, it is vital to pay attention to the cost of achieved scalability on performance in order to avoid “rewarding systems that bring substantial but parallelisable overheads” [118].

---

\(^{11}\) See for example [https://docs.oracle.com/cd/E37231_01/doc.20/e36961/sqlcl.htm#BDCUG125](https://docs.oracle.com/cd/E37231_01/doc.20/e36961/sqlcl.htm#BDCUG125) and [https://dwarehouse.wordpress.com/2012/10/10/greenplum-and-hadoop-hdfs-integration](https://dwarehouse.wordpress.com/2012/10/10/greenplum-and-hadoop-hdfs-integration).

\(^{12}\) See for example [https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions](https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions).
Appendix A

SPARQL Queries Used in Statistical Data Experiment

Q1: all observations that are about Ireland

```sparql
SELECT ?s ?p
WHERE {
  ?s rdf:type qb:Observation .
}
```

Q2: all observations that are about resources equivalent to Ireland (equivalence expressed using skos:exactMatch)

```sparql
SELECT ?s ?o
WHERE {
  ?s rdf:type qb:Observation .
}
```

Q3: observations related between two given datasets. Related here means the two observations have two values related by skos:exactMatch

```sparql
SELECT ?obs1 ?obs2
WHERE {
  ?o skos:exactMatch ?o2 .
}
```
Q4: all observations that are about the first quarter in 2012

```sql
SELECT ?obs1 ?ds1 ?p
WHERE {
  ?ds1 rdf:type qb:DataSet .
}
```

Q5: all datasets with their associated codelists

```sql
WHERE {
  ?ds rdf:type qb:DataSet .
}
```
Appendix B

Bibliometrics SPARQL Queries Used in Evaluating Gagg

c-o-authorship

CONSTRUCT {  
    _:b2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :Coauthorship .  
    _:b2 :author ?auth1 .  
    _:b2 :author ?auth2 .  
    _:b2 <http://example.org/count> ?prop_count .  
}  
WHERE {  
    SELECT ?auth1 ?auth2 (COUNT(DISTINCT ?s) AS ?prop_count)  
    WHERE {  
        ?b0 foaf:name ?name1 .  
        ?b1 foaf:name ?name2 .  
        BIND (IRI(STR(?name1)) AS ?auth1)  
        BIND (IRI(STR(?name2)) AS ?auth2)  
        FILTER (?name1 < ?name2)  
    }  
  }  
}  

Listing B.1: reduced approach

CONSTRUCT {  
    _:b2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :Coauthorship .  
    _:b2 :author ?auth1 .  
}
Listing B.2: fullSparql approach
Listing B.3: 3Sparqls approach

citation

CONSTRUCT{ 
  _:b2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :Citation .
  _:b2 :from ?b0 .
  ?b0 foaf:name ?name1 .
  _:b2 :to ?b1 .
  ?b1 foaf:name ?name2 .
}
Listing B.4: reduced approach
Listing B.5: reduced approach

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX spb: <http://localhost/vocabulary/bench/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX : <http://example.org/>

CONSTRUCT {
  ?b0 foaf:name ?name1 .
  ?b0 <http://example.org/papers> ?papers .
} WHERE {
  SELECT ?b0 ?name1 (COUNT(DISTINCT ?p1) AS ?papers) WHERE{
    ?p1 dc:creator ?b0; a spb:Inproceedings; dct:references ?refs .
    ?b0 foaf:name ?name1 .
    ?b1 foaf:name ?name2 .
  } GROUP BY ?b0 ?name1
}

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX spb: <http://localhost/vocabulary/bench/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX : <http://example.org/>

CONSTRUCT {
  _:b2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :Citation .
  _:b2 :from ?b0 .
  ?b0 foaf:name ?name1 .
  _:b2 :to ?b1 .
  ?b1 foaf:name ?name2 .
  _:b2 <http://example.org/count> ?prop_count .
} WHERE {
  SELECT ?b0 ?name1 ?b1 ?name2 (COUNT(*) AS ?prop_count) WHERE{
    ?p1 dc:creator ?b0; a spb:Inproceedings; dct:references ?refs .
    ?b0 foaf:name ?name1 .
    ?b1 foaf:name ?name2 .
  } GROUP BY ?b0 ?b1 ?name1 ?name2
}

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX spb: <http://localhost/vocabulary/bench/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX : <http://example.org/>

CONSTRUCT {
  ?b1 foaf:name ?name2 .
}
Listing B.6: reduced approach

conf-citation

Listing B.7: reduced approach
Listing B.8: reduced approach

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX spb: <http://localhost/vocabulary/bench/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX : <http://example.org/>

CONSTRUCT {
} WHERE {
  SELECT ?conf1 (COUNT(DISTINCT ?p1) AS ?sub_papers)
  WHERE {
  } GROUP BY ?conf1
}

CONSTRUCT {
  _:b2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> :Citation .
  _:b2 :from ?conf1 .
  _:b2 :to ?conf2 .
  _:b2 <http://example.org/count> ?citation_count .
} WHERE {
  SELECT ?conf1 ?conf2 (COUNT(*) AS ?citation_count)
  WHERE {
  } GROUP BY ?conf1 ?conf2
}
CONSTRUCT {
} WHERE {
  SELECT ?conf1 (COUNT(DISTINCT ?p1) AS ?papers)
  WHERE {
  } GROUP BY ?conf1
}

LISTING B.9: reduced approach
Bibliography


[12] Paolo Atzeni, Christian S Jensen, Giorgio Orsi, Sudha Ram, Letizia Tanca, and Riccardo Torlone. The Relational Model is Dead, SQL is Dead, and I don’t Feel so Good Myself. *ACM SIGMOD Record*, 42(2), 2013.


[33] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S Yu. Graph OLAP: Towards Online Analytical Processing on Graphs. In Eighth International Conference on Data Mining (ICDM’08). IEEE, 2008.


[50] Li Ding, Tim Finin, Yun Peng, Paulo Pinheiro Da Silva, and Deborah L McGuinness. Tracking RDF Graph Provenance Using RDF Molecules. In The 4th International Semantic Web Conference (Poster), 2005.


[95] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In OSDI, 2012.


[99] Kisung Lee and Ling Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. VLDB Endowment, 6(14), September 2013.


[177] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph Cube: On Warehousing and OLAP Multidimensional Networks. In ACM SIGMOD International Conference on Management of Data, SIGMOD ’11. ACM, 2011.