ABSTRACT

KIM, HYEONGSIK. Semantics-Oriented Optimization Techniques for Large Semantic Network Processing. (Under the direction of Kemafor Ogan.)

Large Semantic Networks are becoming increasingly pervasive in recent data analysis scenarios because of the need to integrate and query a broad range of data sources. A semantic network is a labeled directed graph where vertices and edges represent concepts and semantic relationships between concepts respectively. Many semantic networks available on the Web are expressed using the Resource Description Framework (RDF), which is a standard data model for the Semantic Web. Semantic networks are also often accompanied by ontologies which describe specifications of concepts and relations in a domain using ontology languages such as RDF Schema (RDFS) and Web Ontology Language (OWL). The standardized concepts and relations in ontologies allow interlinking different networks, and therefore large-scale networks begin to emerge as many ontologies are publicly available on the Web.

Semantic networks can be used for answering various types of queries, but the most common workload is processing graph pattern matching query that seeks subgraphs that match patterns explicitly stated in queries. However, semantic networks also represent entailed structures that are not explicitly described in queries. Therefore, another common workload is processing ontological query expressed in terms of ontological descriptions to capture both explicit and entailed structures in networks. The entailed structures can be computed using forward/backward-chaining techniques, but they often do not scale well with respect to the number of rules and the volume of datasets. More recently, some fragments of OWL (e.g., OWL2 QL) support scalable inferencing by reformulating queries into unions of conjunctive queries (UCQs or unions of graph patterns) based on ontological axioms. While the reformulation would allow leveraging database systems to process ontological queries in a scalable manner, in reality UCQs are often composed of many conjunctive patterns connected by unions, which poses challenges for an efficient evaluation of UCQs. It is therefore crucial to optimize the evaluation of UCQs to support efficient query answering over large semantic networks.

For optimizing the evaluations of UCQs, many different strategies can be considered based on which processing and storage models are employed. Most existing models employ fine-grained storage models, which store semantic networks edgewise. Consequently, query processing involves incremental assembling of matching edges using many joins and unions. However, processing many operations often involves a considerable amount of overhead. In particular, when distributed processing platforms such as Hadoop are employed to process large-scale networks, the cost of joins becomes prohibitively expensive because each operation leads to re-partitions of intermediate data, causing a substantial amount of disk and network I/Os.

From our observation, these challenges are stemmed from the fact that most existing models are semantic-oblivious because they do not properly capture the semantic perspective of semantic networks
such as (i) semantic equivalence between subexpressions of graph patterns, (ii) semantic context of resource, and (iii) formal semantics of properties on concepts and definitions in accompanied ontologies. The goal of this dissertation is, therefore, developing *semantic optimization techniques* which exploit semantic properties of RDF queries and data for optimizing query processing over semantic networks. In particular, the techniques introduced in this dissertation focus on (i) optimizing query expressions to eliminate redundant evaluation of “contents-equivalent” conjunctive subexpressions using rewriting techniques (ii) developing type-aware query processing and storage models which allow for eliminating redundant assertions by leveraging ontological axioms and entailment rules, and (iii) improving the evaluation of UCQs by sharing the evaluation of contents-equivalent disjunctive subexpressions over the type-aware model.

We implement the type-aware model along with the proposed optimizations using Apache Tez, which is a distributed processing framework running on Apache Yarn. Empirical evaluation demonstrates that our approach is highly effective, which is up to 25X faster than existing approaches.
© Copyright 2017 by HyeongSik Kim

All Rights Reserved
Semantics-Oriented Optimization Techniques for Large Semantic Network Processing

by

HyeongSik Kim

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

Rada Chirkova

Xipeng Shen

Ranga Vatsavai

Kemafor Ogan
Chair of Advisory Committee
DEDICATION

To my parents.
BIOGRAPHY

HyeongSik Kim was born to a loving couple, JiChul Kim and BokHee Byeon. He received his Bachelor of Engineering degree (Digital Information Engineering) in 2009 from Hankuk University of Foreign Studies. After working as an intern in Samsung Electronics, he chose to continue his graduate studies. His research interests include developing optimization techniques for scalable processing of Semantic Web data and building scalable query processing frameworks on cloud platforms. During his PhD program, he had internship experiences from Bosch, USA (Summer 2016).
ACKNOWLEDGEMENTS

I would never have been able to finish my dissertation without the guidance of my committee members, assistant from lab peers, and support from my family. First and foremost, I would like to express my sincere gratitude to my advisor, Kemafor Anyanwu, for the continuous support of my Ph.D study and related research. Without her guidance and persistent help, this dissertation would not have been possible. I also thank Rada Chirkova, Xipeng Shen, and Ranga Vatsavai for serving on my doctoral committee and providing valuable feedback on my dissertation research. I am grateful to my internship mentor, DooSoon Kim (Bosch USA) for their continued support and guidance that has extended beyond summer internships. I also would like to thank Aaron Peeler and his team at the Virtual Computing Lab (VCL), North Carolina State University for their help and support in reserving large-scale Hadoop clusters for my study. Heartfelt gratitude goes to all the helpful staff in the Department of Computer Science at the North Carolina State University.

I would like to acknowledge the support of my peers, both present and past members of the Semantic Computing Research Lab (COUL) - Haizhou Fu, Sidan Gao, Padmashree Ravindra, Seokyong Hong, Sahibi Miranshah, Avimanyu Mukhopadhyay, Shalki Shrivastava, Ruth Okoilu, and Abhisha Bhattacharyya. I would like to specially thank Padmashree Ravindra and Seokyong Hong for enlightening me the first glance of research and helping me to develop my background. I would also like to extend my thanks to Haizhou Fu and Sidan Gao for providing a friendly and cooperative atmosphere at work and insightful comments on my work. My sincere thanks also goes to Holly J. L. Kim and Eugene M. Kim for the support they have lent me during my stay at San Jose.

Last but not the least, I would like to extend my deepest gratitude to my family, in particular my parents, JiChul Kim and Bokhee Byeon, and sister, SeulA Kim, for their support during my dissertation and throughout my degree as well as my career in general. Without them I would not be where I am today. Thank you.

This research was supported in part by the National Science Foundation under Grant No. IIS-0915865, IIS-1218277, and CNS-1526113.
# TABLE OF CONTENTS

List of Tables .................................................................................................................. vii

List of Figures ...................................................................................................................... viii

Chapter 1  Introduction ........................................................................................................ 1

1.1 Key Components ........................................................................................................ 4

1.1.1 Optimizing the Processing of Queries with Semantically Related Subexpressions .... 4

1.1.2 Automatic Semantic Typing Scheme for Semantics-Oriented RDF Storage Model on Hadoop ..................................................................................................................... 5

1.1.3 Optimizing Processing of Complex Queries on RDF data Using SemStorm ........... 5

1.2 Related Work ............................................................................................................. 6

1.3 Outline ....................................................................................................................... 8

Chapter 2  Preliminaries .................................................................................................... 9

2.1 Resource Description Framework Schema (RDFS) .................................................... 9

2.2 RDFS Entailment .................................................................................................... 10

2.3 SPARQL Protocol and RDF Query Language (SPARQL) ............................................. 12

2.4 The MapReduce Framework .................................................................................... 12

2.4.1 MapReduce ......................................................................................................... 12

2.4.2 Hadoop Distributed File System (HDFS) ............................................................. 13

2.5 SPARQL Query Processing on MapReduce ................................................................ 14

2.5.1 Graph Pattern Matching Using Relational Algebra ............................................. 14

2.5.2 Nested Triplegroup Algebra and Data Model (NTGA) ........................................... 15

Chapter 3  Optimizing the Processing of Queries with Semantically Related Subexpressions 17

3.1 Motivation .................................................................................................................. 17

3.1.1 Contributions ..................................................................................................... 18

3.1.2 Related Work ..................................................................................................... 18

3.2 Foundations .............................................................................................................. 19

3.2.1 Well-formed, Ambiguous, and Perfect Triplegroups ........................................... 19

3.2.2 Triplegroup Cloning .......................................................................................... 20

3.3 Empirical Evaluation ............................................................................................... 21

3.3.1 Testbed Setup .................................................................................................... 22

3.3.2 Evaluation Results ............................................................................................ 23

3.4 Chapter Summary ..................................................................................................... 25

Chapter 4  Automatic Semantic Typing Scheme for Semantics-oriented RDF Storage Model on Hadoop ................................................................. 26

4.1 Motivation ................................................................................................................ 26

4.1.1 Contributions .................................................................................................... 28

4.1.2 Related Work .................................................................................................... 29

4.2 Type-Based Optimization of RDF Graph Pattern Queries ........................................... 29

4.2.1 R-Type: A Typing Model for RDF ..................................................................... 30
4.2.2 Autonomic R-Type Model Generation and Storage on Hadoop 30
4.2.3 Implementation of R-Type Model Generation 32
4.3 Query Processing on SemStorm 34
4.3.1 Query Compilation 34
4.3.2 Query Execution 36
4.4 Semantic Optimization Using Ontological Axioms 37
4.4.1 Optimizing SemStorm’s Physical Model Using Triple Dematerialization 37
4.4.2 Eliminating Redundant Joins from R-Type Expressions 39
4.5 Empirical Evaluation 43
4.5.1 Testbed Setup 43
4.5.2 Evaluation Results 46
4.6 Chapter Summary 50

Chapter 5 Optimizing Processing of Complex Queries on RDF Data Using SemStorm 51
5.1 Motivation 51
5.1.1 Related Work 54
5.2 Optimization Strategies for Processing UCQs in Distribute Contexts 54
5.2.1 Increasing the Selectivity of Graph Patterns in UCQs 54
5.2.2 Rewriting Strategies for Optimizing the execution of UCQs 56
5.3 Optimizing UCQs Using Type-based Semantic Reformulation 59
5.3.1 Rewriting Techniques for Eliminating Chain-Joins via Inverting 60
5.3.2 Extending R-Type Query Processing Model for UCQs 61
5.4 Empirical Evaluation 63
5.4.1 Testbed Setup 63
5.4.2 Evaluation Results 65
5.5 Chapter Summary 71

Chapter 6 Conclusion and Future Work 72
6.1 Overview of Dissertation 72
6.2 Future Directions 73

References 75
LIST OF TABLES

Table 2.1  RDF statement and RDFS constraints  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
Table 4.1  Dataset characteristics  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
Table 5.1  Dataset characteristics  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
Table 5.2  Storage requirement and preprocessing time of the benchmarked approaches  . 63
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An example (a) semantic network and (b) ontology describing an academic domain, and (c) a set of RDF triples corresponding to the example network and ontology. Labels over edges are abbreviated due to space constraints, e.g., rdf:type is represented as :t.</td>
</tr>
<tr>
<td>1.2</td>
<td>(a) An example graph pattern matching query Q and (b) an relational expression for the query Q.</td>
</tr>
<tr>
<td>1.3</td>
<td>A categorization of related work.</td>
</tr>
<tr>
<td>2.1</td>
<td>(a) An example SPARQL query Q, (b) a graph representation of the query Q, and (c) an relational expression for the query Q.</td>
</tr>
<tr>
<td>2.2</td>
<td>An overview of MapReduce job execution.</td>
</tr>
<tr>
<td>2.3</td>
<td>(a) a set of triplegroups corresponding to example triples and (b) content-equivalence between a triplegroup $tg_5$ and n-tuples $tup_1$ and $tup_2$.</td>
</tr>
<tr>
<td>2.4</td>
<td>(a) A relational query execution plan and (b) a NTGA-based query execution plan for query Q.</td>
</tr>
<tr>
<td>3.1</td>
<td>(a) An example query $RQ$ and (b) a query execution plan for $RQ$ on MapReduce.</td>
</tr>
<tr>
<td>3.2</td>
<td>Cloning in the context of ambiguous triplegroups.</td>
</tr>
<tr>
<td>3.3</td>
<td>Comparison of MR execution time of three approaches (Pig-Def, NTGA, and SHARD) for dq4 (BSBM-250k).</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparison of the execution time, the number of HDFS read/written for varying the number of DupPs ((a)∼(c)) and the frequency of DupPs in a query ((d)∼(e)).</td>
</tr>
<tr>
<td>3.5</td>
<td>Scalability study of scan-sharing approach for query dq4 with increasing sizes of RDF graphs.</td>
</tr>
<tr>
<td>4.1</td>
<td>(a) Graph representation of graph pattern query Q that retrieves faculties who lecture courses (b) star subgraphs (triplegroups).</td>
</tr>
<tr>
<td>4.2</td>
<td>A physical representation of typed triplegroups using polymorphic-typed file organization and file format.</td>
</tr>
<tr>
<td>4.3</td>
<td>Pre-processing triples to construct triplegroups using RAPID+ on SemStorm.</td>
</tr>
<tr>
<td>4.4</td>
<td>Retrieving RDF triples matching queries from SemStorm.</td>
</tr>
<tr>
<td>4.5</td>
<td>Dematerializing a D-inferable triple (:prs1, rdf:type, :Faculty) in a triplegroup $tg_1$ based on a domain constraint (:l, rdfs:domain, :crs1) and RDFS entailment rule (rdfs2).</td>
</tr>
<tr>
<td>4.6</td>
<td>An overall processing flow of the query Q over the R-Type model with semantic optimization techniques.</td>
</tr>
<tr>
<td>4.7</td>
<td>An overall architecture of SemStorm on Hadoop 2.</td>
</tr>
<tr>
<td>4.8</td>
<td>Comparisons of queries without rdf:type triple patterns using LUBM Univ 20k dataset on 80-node cluster: (a) high-selectivity queries and (b) low-selectivity queries.</td>
</tr>
<tr>
<td>4.9</td>
<td>Comparisons on 20-node cluster using DBSPB-200 dataset: (a) query execution times, (b) HDFS read bytes, and (c) aggregated execution times of queries without rdf:type triple patterns.</td>
</tr>
</tbody>
</table>
Figure 4.10  Answering negative queries (a) LUBM (b) DBPSB ................................. 49
Figure 4.11  Evaluating queries with rdf:type triple patterns using LUBM datasets (Univ-20k, 80-node cluster) ................................. 50
Figure 4.12  A performance comparison of queries with multiple (non-)inferable rdf:type triple patterns (DBSPB-200, 20-node cluster) ................................. 50

Figure 5.1  Rewriting two star pattern stp_1 and stp_2 ................................. 52
Figure 5.2  (a) An execution plan for a reformulation of the triple pattern tp, (?s rdf:type ub:Student) and (b) the rewriting process that generates more selective plans. ................................. 56
Figure 5.3  Execution plans for a graph pattern with (a) disjunctive predicates and (b) unions of reformulated triple patterns. (c) An reformulated query using multi-query optimization and (d) an execution plan of the query ................................. 57
Figure 5.4  An overall flow of semantic optimizations for range-invertable triples: (a) data and (b) query rewriting process. ................................. 59
Figure 5.5  Interpreting ontological query gp as (a) unions of FOL formulas and (b) disjunctions of MSO formulas. ................................. 62
Figure 5.6  Execution time of queries with a varying number of union branches: (a) high and (b) low selectivities (unit: s) ................................. 66
Figure 5.7  Execution time of queries with a varying size of common expressions: (a) high and (b) low selectivities (unit: s) ................................. 68
Figure 5.8  HDFS read of queries with a varying size of common expressions: (a) high and (b) low selectivities (unit: GB) ................................. 68
Figure 5.9  Execution time of queries with a varying number of rdfs:range axioms: (a) high and (b) low selectivities (unit: s) ................................. 71
Chapter 1

Introduction

Recently large Semantic Networks are becoming increasingly common in data analysis scenarios because of the need to integrate and query a broad range of data sources. For example, public health and biomedicine communities have actively built and used large semantic networks such as Knowledge Base of BioMedicine (KABOB) [56], Uniprot [20], BioPortal [70]. These semantic networks are essentially labeled directed graphs where vertices and edges represent concepts and semantic relationships between concepts, respectively. Fig. 1.1(a) shows an example semantic network describing resources in academic domains such as faculties (e.g., :prs1, :prs2, and :prs3) and courses they are teaching (e.g., :crs1). Many semantic networks available on the Web are often expressed using the Resource Description Framework (RDF) [17], which is a standard data model for the Semantic Web. RDF offers a common framework for describing knowledge using a triple (Subject, Property, Object) which asserts that a Subject has a Property whose value is denoted by the Object. For example, the triple (:prs1, :name, “aaron”) asserts that the resource :prs1 has a named Property :name whose Object is the literal “aaron”. The standardization based on RDF allows different semantic networks to be automatically interconnected, which leads to the emergence of large semantic networks. For example, Uniprot is a database of protein sequences which consists of loosely interconnected resources from many different datasets such as genomes, enzymes, publications, etc, and the number of RDF triples in the database recently reaches approx. 17B.

In the context of semantic network processing represented in RDF, two classes of queries can be considered.
1. The first class of queries called graph pattern matching queries is ‘shallow’ in that they focus on finding exact structural matches to a given query pattern. Fig. 1.2(a) shows an example graph pattern matching query that retrieves a list of faculties and the courses that they teach. Variable are denoted by leading ’?’, which are mapped to resources in a semantic network, e.g., valid bindings for the variables ?prs and ?crs in Fig. 1.2(a) are :Prs1 and :Crs1 respectively.
2. The second class of queries is ontological queries. Many real-life semantic networks are accompa-
Figure 1.1: An example (a) semantic network and (b) ontology describing an academic domain, and (c) a set of RDF triples corresponding to the example network and ontology. Labels over edges are abbreviated due to space constraints, e.g., \texttt{rdf:type} is represented as \texttt{:t}.

Ned by ontologies described using standard ontology languages such as RDF Schema (RDFS) \cite{15} and Web Ontology Language (OWL) \cite{14}. These ontologies provide specifications of concepts and axioms that allow for deriving implicit knowledge represented in networks. For example, we can specify a domain relationship using a \texttt{rdfs:domain} property in RDFS, e.g., \texttt{(':lectures, rdfs:domain, :Faculty')}, in Fig. 1.1(b) denotes that a resource that describes by a property \texttt{:'lectures} is an instance of the class \texttt{:'Faculty}. Based on this relationship, we can infer a new triple \texttt{('prs1, rdf:type, :Faculty')}, if a network contains a triple \texttt{('prs1, :lectures, :Database')}. Ontological queries are ‘deep’ in that they are expressed in terms of ontological descriptions to find not only explicit structures, but also implied structures which may not match exactly to a given query pattern. For example, if an ontological query includes a pattern \texttt{('?prs, rdf:type, :Faculty')}, it will also find implicit structures such as \texttt{('prs1, :lectures, :crs1')}, based on the domain relationship discussed above.

To support ontological query processing, many RDF storage systems \cite{6, 7, 12, 33, 47, 82} employ the strategy of pre-materializing all implied structures using techniques such as \texttt{forward-chaining}. The advantage of this approach is that ontological query can be processed in the same way as traditional query answering since all implied data has became explicit. However, this approach is often impractical for large semantic networks because the materialization process is computationally expensive.
and involves orders of magnitude blow-up in data size. Other systems [69, 80] employ backward-chaining which computes only the relevant part of the closure in a dynamic way. However, these algorithms tend to be inherently sequential making them difficult to parallelize and scale up for large semantic networks. But more recently, some fragments of OWL (e.g., OWL2 QL) support scalable inferencing over large semantic networks by rewriting queries into unions of conjunctive queries (UCQs or unions of graph patterns) based on ontological axioms [26, 23, 36, 52]. As a simple example, the pattern (?prs, rdf:type, :Faculty) can be rewritten into a union of (?prs, rdf:type, :Faculty) and (?prs, :lectures, ?x) to explicitly find both implicit and explicit structures.

While UCQs can be evaluated using existing RDF databases, the number of graph patterns in UCQs is often very high, which may pose challenges for efficient evaluation of UCQs. For example, the ontology in Uniprot consists of over 1M classes and 1M rdfs:subclassOf statements. Therefore, a trivial ontological query on Uniprot that involves the class Taxon could easily exceed 800 disjuncts, with each disjunct being a multi-join graph pattern. It is therefore crucial to optimize the evaluation of UCQs to support efficient query answering over large semantic networks.

There are different strategies for processing graph pattern queries based on which processing and storage models are employed. However, most existing models employ fine-grained storage models, which store RDF datasets edgewise [5, 19, 33, 34, 41, 43, 44, 54, 61, 63]. Consequently, query processing requires many join operations to incrementally assemble different edges of matching subgraphs, which involves a considerable amount of overhead. Assuming that example RDF triples are vertically partitioned on Properties [19], the query Q can be translated into an expression as shown in Fig. 1.2(b). Note that a number of joins in this expression is 7 because the query consists of a graph pattern with 8 edges. In reality, several join operations are typical in graph pattern matching queries [21]. For this reason, graph pattern matching query processing often requires a significant amount of computing resources and processing time. In particular, the costs of joins are significantly expensive on distributed platforms such as MapReduce [31] because each operation is associated with a substantial amount of disk and network I/Os. To mitigate such costs, various techniques have been investigated such as multi-indexing schemes [61], co-partitioning techniques [34], structure-aware partitioning techniques [43, 54].
and sideways information passing [61].

Note that these techniques primarily focus on exploiting structural properties of the RDF data model such as co-partitioning subgraphs based on common nodes/edges. However, they do not consider leveraging semantic properties of the RDF data model such as (i) semantic equivalence between subexpressions of graph patterns and (ii) formal semantics of properties with respect to concepts and definitions in accompanied ontologies. The goal of this dissertation is therefore developing semantic optimization techniques which exploit semantic properties of RDF queries and data for optimizing graph pattern processing on MapReduce.

1.1 Key Components

In this dissertation, we present three components focusing on (i) optimizing query expressions to eliminate redundant evaluation of “contents-equivalent” conjunctive subexpressions using rewriting techniques (ii) developing a type-aware storage model which allows for eliminating redundant assertions by leveraging ontological axioms and entailment rules, and (iii) improving the evaluation of UCQs by sharing the evaluation of contents-equivalent disjunctive subexpressions over the type-aware storage model.

1.1.1 Optimizing the Processing of Queries with Semantically Related Subexpressions

Most conjunctive graph pattern matching queries contain semantically related subexpressions, e.g., common subexpressions across multiple star patterns. Existing approaches often represent the subexpressions based on the first-order relational expressions and evaluate the expressions independently, which results in significant performance overhead. However, the expressions can be reformulated into semantically equivalent second-order expressions using Nested TripleGroup Data Model and Algebra (NTGA). NTGA employs a triplegroup (or a star subgraph) as a first-class citizen where each group is a set of triples that have the same Subject resource. These triplegroups can be considered as candidate matches for star patterns in queries, therefore an NTGA-based query plan includes operators that scan all triplegroups and select only triplegroups that match (one of) star patterns. Such a query plan allows for evaluating all star subgraph patterns concurrently, which provides opportunities for processing all semantically related subexpressions in different star patterns. However, this model introduces challenges in query executions. In particular, an interpretation of semantically related subexpressions using NTGA needs to handle ambiguities. Because multiple star graph patterns include common subexpressions which need to be evaluated concurrently, a triplegroup can potentially match multiple star patterns, creating ambiguities during the matching process.

Research Questions: What are the possible evaluation strategies to share evaluations of semantically related subexpressions? How can we effectively address ambiguities in NTGA?
**Proposed Solution:** This component proposes to extend NTGA operators to deal with ambiguities, which allows for avoiding redundant evaluation of semantically related subexpressions. These operations focus on transforming an single subgraph (triplegroup) into matches of multiple subexpressions, thereby avoiding multiple scans and multiple query execution paths for semantically related subexpressions.

### 1.1.2 Automatic Semantic Typing Scheme for Semantics-Oriented RDF Storage Model on Hadoop

Typing in databases is fundamental to query optimization, e.g., for compile-time semantic analysis of query correctness and pruning of state-space during query execution. Typing is also employed in *Semantic Query Optimization (SQO)* where it can be combined with integrity constraints for use in rewriting queries into more efficient ones by eliminating redundant expressions, e.g., eliminating joins. However, when dealing with semi-structured data that often lack strongly typed models (and in some cases even schemas), enabling similar type-based optimizations is challenging. While this has been considered for XML data, the problem remains open in the context of RDF. Given the growing popularity of RDF and the challenges that still remain with respect to query processing in the large, the investigation of type-based query optimization techniques for RDF is likely to be a worthwhile research direction.

**Research Questions:** What kind of typing models would be appropriate for RDF data? How can types be automatically induced from a triple model and schema to avoid the need for users to explicitly define them? What is an appropriate storage organization and physical representation for types?

**Proposed Solution:** To address the questions, this component proposes a two-pronged approach to enabling type-based optimization for Big RDF query processing. First is an *Autonomic Typing Model for RDF called R-Typing* that (i) autonomically induces a strongly typed (R-Type) model for RDF and rewrites SPARQL graph pattern queries into expressions over R-Types to achieve state-space pruning, and (ii) a distributed storage and indexing scheme for R-Typed Models called *SemStorm*, which introduces a *polymorphic nested column-oriented file organization and format* for Hadoop Distributed File System (HDFS). SemStorm enables efficient index-based access to query relevant data from HDFS. Second, we propose semantic optimization techniques which exploit ontological axioms in ontologies that often accompany RDF data, in a manner analogous to integrity constraints. This enables optimizations of both SemStorm’s physical storage models (eliminating ‘redundantly stored data’) and queries (eliminating ‘redundant’ subexpressions).

### 1.1.3 Optimizing Processing of Complex Queries on RDF data Using SemStorm

Beyond structural matching by graph pattern matching queries, RDF datasets also can be queried using other complex form of queries such as ontological queries. Ontological queries are often translated
into unions of conjunctive queries (UCQs), which are generated by rewriting conjunctive queries with respect to ontological constraints. However, processing UCQs is orders of magnitude more challenging than regular graph pattern queries because the number of conjunctive queries tends to be exponential. It is particularly difficult to evaluate such queries on distributed platforms because many union branches where each of them contains joins are translated into a significant number of operators. Such a large number of operators often lead to a large number of data re-partitions of intermediate data, which obviously involves a huge amount of disk and network I/Os.

**Research Questions:** How can we translate UCQs into more optimized query plans that do not require a large number of data re-partitions? What are the possible execution strategies to optimize such plans?

**Proposed Solution:** A worthwhile observation is that UCQs tend to contain many semantically related subexpressions across union branches, which implies that the evaluation of such expressions can be shared over the semantics-oriented storage model. In this context, we focus on developing alternative query rewritings that can produce more efficient expressions in the context of distributed environments. We adapt two classes of techniques to the context of a strong type system for RDF data: (i) **Union Pushdown** - an algebraic method for reformulating disjunctive queries and (ii) **Multi Query Optimization** - which attempts to achieve shared execution of common subexpressions across subqueries in UCQs. We also integrate such techniques with semantic optimizations that utilize ontological axioms to eliminate common chain-joins in UCQs, which further allows reducing the re-partitions of intermediate data. Comprehensive experiments conducted on synthetic benchmark datasets with customized query sets show that our reformulation techniques with semantic optimizations are highly effective, which is up to 25X faster than existing approaches.

### 1.2 Related Work

**Semantic and Type-based Query Optimization.** The idea of semantic query optimization (SQO) is to exploit integrity constraints such as inclusion dependencies and functional dependencies or semantic knowledge from applications to reformulate a query into one which is less expensive but semantically equivalent. In relational systems or logic-based models such as Datalog, several different types of SQO have been studied to enhance literals [45], eliminate literals [84, 49, 50, 73, 74], introduce joins [29], or reduce scans [49, 74]. However, such techniques have rarely been considered for optimizing RDF query processing because these SQOs assumes strongly typed data models such as the relational model which is quite different for graph models like RDF. [59, 22] presents a theoretical study of type inferencing and checking with respect to semi-structured data models. The authors conclude that while in functional programming the problem of type checking is fairly straightforward, both type checking and type inferencing for queries over semi-structured data are NP-Hard. [58] showed that integrating type information with other kinds of constraints could allow non-trivial constraint-based query rewritings.
that are not possible when each type is considered separately.

**Queries with Common Subexpressions.** Many efforts have been made to exploit similar or related subexpressions for optimization. In particular, it has been extensively studied in the context of optimizing multiple related queries, i.e., multi-query optimizations (MQO) in [68, 87]. MQO is also studied in the context of optimizing multiple graph pattern matching queries [53]. In addition, there are also a number of studies that identify similar workflows and share the execution on MapReduce. For example, MRShare [62] identifies different types of sharing opportunities by analyzing MR job workflows generated from a set of queries and shares the execution or the output of Map/Reduce phases to reduce the cost of execution. Recently, these techniques have been studied in the context of intra-query optimizations, which leverage related subexpressions in subqueries. Cascade-style optimizers in Scope [75] is extended to employ such optimizations and select globally optimal plans based on its cost model. YSmart [55] leverages correlations between subqueries to reduce the cost of query execution on MapReduce. It employs a correlation-aware SQL-to-MapReduce translator that identifies correlated subplans and merges them based on a set of rewriting rules to minimize the number of MR jobs. However, they only focus on query transformation for optimization without considering a transformation of intermediate data to share the evaluation of related subexpressions.

**Storage Models for RDF Data.** There are two dimensions that can be used to classify existing storage
models for RDF data model, i.e. *triple-based models* [61, 44, 19, 41, 40, 5, 83, 33, 72] vs. *subgraph-based models* [54, 43, 88, 86, 65]. The former uses a triple for the units of storage and retrieval while the latter uses coarser-grained units of storage and retrieval such as subgraphs. Whereas the triple-based models require a join operation to add each new triple to a subgraph, the subgraph-based models have the advantage of requiring fewer join operations during query processing since triples have already been assembled into coarser-grained units. Another perspective of storage models is the issue of whether they are *typed* [88, 86] or *untyped* [65, 54, 43]. Typed storage models partition data based on types, labels, or signatures of data units so that units with the same type (or label/signatures) can easily be differentiated. The advantage of such differentiated storage models is that graph patterns can easily be matched without processing triples whose types are not included in a query. On the other hand, untyped storage models which do not partition data do not enable that level of discrimination during query processing. However, they often rely on specialized access methods such as indexes to achieve selective access based on either the subject or the object of triples.

**Ontological Query Processing.** Query rewriting algorithms for ontological queries has been extensively studied for RDFS or OWL2 QL [78, 27, 36]. However, such algorithms often produce an exponential size of the rewritings [27], i.e., generated UCQs often contain hundreds or thousands of subqueries in reality, which are too big to be evaluated efficiently on existing database systems. To mitigate this complexity issue, much effort has been put into the development of optimization techniques that produces more succinct queries [36] or rewrites queries into other expressive languages such as Datalog [67, 32]. In recent years, a few authors have begun to investigate techniques for optimizing the execution of UCQs as orthogonal approaches for handling the complexity issue. In [25], ontological queries are decomposed into different combinations of (possibly overlapping) subqueries, producing a set of covers for queries. Since the queries reformulated from these covers have different query processing costs, efficient one is selected and evaluated using a greedy cost-based algorithm. While such techniques can be effective for traditional query processing systems, it is possible that selected plans would not be plans that produce optimized MapReduce job flows for processing ontological queries.

### 1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 covers backgrounds for the research presented in this dissertation. Chapter 3 discusses optimization techniques for queries with semantically related subexpressions. Chapter 4 presents our automatic semantic typing scheme and type-aware storage models on Hadoop. Optimization of complex query answering over the type-aware storage model is discussed in Chapter 5 and Chapter 6 includes the conclusion and future work.
Chapter 2

Preliminaries

2.1 Resource Description Framework Schema (RDFS)

Resource Description Framework Schema (RDF Schema/RDFS)\(^1\) is a set of standardized vocabularies
for describing concepts and semantic relationships between the concepts. RDFS represents concepts
using special vocabularies \textit{rdfs:Class} and \textit{rdf:type}. For example, there are three classes describing
corcepts \textit{Person}, \textit{Course}, and \textit{Faculty} in Fig. 1.1(b), which can be represented by adding new schema
triples such as (:Person, rdf:type, rdfs:Class), (:Course, rdf:type, rdfs:Class), and (:Faculty, rdf:type, rdfs:Class).

Resources can be categorized into classes and resources that belong to a class are called as instances
of that class. These relationships can be described using \textit{rdf:type}. For example, (:prs1, rdf:type,
:Faculty) asserts that :prs1 is an instance of class :Faculty. Note that RDFS allows resources to
have multiple classifications without necessarily conforming descriptions of classes they belong to. For
example, \textit{rdf:type} statements in Fig. 1.1(b) assert that :prs1 is an instance of class :Faculty as well as
:Person.

RDFS also provides two special properties \textit{rdfs:domain} and \textit{rdfs:range} to define associations
between classes and properties. \textit{rdfs:domain} associates a class with a Property’s Subject. For example,
(:lectures, rdfs:domain, :Faculty) denotes that the subject described using a property \textit{lectures}
is an instance of the class :Faculty. Similarly, \textit{rdfs:range} associates a class with a Property’s Object.
For instance, (affWith, rdfs:range, xsd:String) expresses that the object of property \textit{takeCourse}
are represented as String Data Types defined in XML Schema Language\(^2\).

Class hierarchy also can be expressed using a special property called \textit{rdfs:subClassOf}, e.g.,
(:Faculty, rdfs:subClassOf, :Person) represents that :Faculty is a subclass of :Person. Likewise,
property hierarchy can be represented using a property \textit{rdfs:subProperty}, e.g., (:hasMasterDegreeOf,

\(^1\)http://www.w3.org/TR/rdf-schema
\(^2\)https://www.w3.org/TR/xmlschema11-1
Q. SELECT * WHERE {
?prs :name ?n1 .
?prs rdf:type :Faculty .

?crs :hpage ?h .
?crs :mbox ?m .
?crs :name ?n2 .
?crs rdf:type :Course .
}

stp1 = \{a, n, l, t\_f\}

stp2 = \{h, m, n, t\_c\}

(stp1 ⊙ (?:affWith ⋈(?prs = ?prs):name ⋈ (?prs = ?prs) ... ⋈ ...)
(stp2 ⊙ (?:hpage ⋈ (?crs = ?crs):mbox ⋈ (?crs = ?crs) ... ⋈ ...))

Figure 2.1: (a) An example SPARQL query Q, (b) a graph representation of the query Q, and (c) an
relational expression for the query Q.

rdfs:subPropertyOf, :hasDegreeOf) expresses that :hasMasterDegreeOf is a subproperty of :hasDegreeOf.

2.2 RDFS Entailment

RDFS provides a set of entailment rules that allow infer implicit information from schema and data.
Among RDFS entailment patterns, we briefly cover some of core ones used in next chapters below.
Variables are denoted with a leading question mark such as ?a.

1. Rule rdfs2: rdfs2 infers a new rdf:type triple (a triple with a property rdf:type) through rdfs:domain
constraints.

\[
\begin{align*}
\langle ?a, rdfs:domain, ?x \rangle & \quad ?u, ?a, ?x \rangle \\
\langle ?u, rdfs:type, ?x \rangle
\end{align*}
\]

For example, given triples (:takeCourse, rdfs:domain, :Student) and (:prs1, :takeCourse, :Database),
a new triple (:prs1, rdf:type, :Student) can be inferred based on rdfs2.

2. Rule rdfs3: rdfs3 derives a new rdf:type triple through rdfs:range constraints.

\[
\begin{align*}
\langle ?a, rdfs:range, ?x \rangle & \quad \langle ?x, ?a, ?u \rangle \\
\langle ?u, rdfs:type, ?x \rangle
\end{align*}
\]

Using rdfs3, a new triple (:Database, rdf:type, :Course) can be inferred from triples (:takeCourse,
rdfs:range, :Course) and (:prs1, :takeCourse, :Database).

These rules can be used for deriving facts implicitly represented in networks and this can be achieved
using a number of different techniques. We borrow some of notions in [25] with slight modifications to
explain these techniques as follows.
Table 2.1: RDF statement and RDFS constraints

<table>
<thead>
<tr>
<th>RDF statement</th>
<th>Triple</th>
<th>RDFS statement</th>
<th>Triple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class assertion (rdf:type)</td>
<td>s τ o</td>
<td>Subclass (rdfs:subClassOf)</td>
<td>s ⊆sc o</td>
</tr>
<tr>
<td>Property assertion</td>
<td>s p o</td>
<td>Subproperty (rdfs:subPropertyOf)</td>
<td>s ⊆sp o</td>
</tr>
<tr>
<td>Domain type (rdfs:domain)</td>
<td></td>
<td></td>
<td>s ⊆d o</td>
</tr>
<tr>
<td>Range type (rdfs:range)</td>
<td></td>
<td></td>
<td>s ⊆r o</td>
</tr>
</tbody>
</table>

Let RDF knowledgebase \(kb\) be a tuple \(⟨S, D⟩\) where \(S\) and \(D\) represent a set of schema triples and instance triples, respectively. Schema triples are a set of statements that conform the form in the right side of Table 2.1, which shows templates of the triples that use meta classes/properties defined in RDF Schema (RDFS) [15]. Instance triples are a set of statements that follow the form in the left side of the Table 2.1, which lists a general form of RDF triples. Answering a query \(q\) over \(kb\) involves the process of finding explicit matches over \(D\) but also deriving implicit information using the process called RDFS entailment [15]. This process can be achieved using the following techniques.

- **Materialization.** The techniques under this category [81, 79, 35] repeatedly applies entailment rules over \(kb\) in a forward-chaining manner, materializing the closure of \(kb\). The evaluation of query \(q\) over the materialization produces complete answers, which can be represented as \(q(Materialize(kb)) = q(kb^∞)\) where Materialize and \(kb^∞\) denote the algorithm and the closure of \(kb\) respectively.

- **Query Reformulation.** This type of techniques [80, 35, 25] reformulates \(q\) by applying a reformulation algorithm over \(kb\) in a backward-chaining manner, which can be denoted as \(Reformulate(q, kb)\). The algorithm returns a reformulated query, \(q^{ref}\), which is composed of a union of basic graph pattern queries \(Reformulate(q, kb) = q^{ref}\). Note that the evaluation of the reformulated query over non-materialized \(kb\) returns complete answers, which are equal to the ones using the materialization technique, i.e., \(q(kb^∞) = q^{ref}(kb)\).

- **Hybrid Approach.** The authors in [78] observe that closure computations of (i) rdf:type statements and (ii) other statements dependent on the rdf:type statements are major overheads of the materialization approach. To relieve this overhead, the authors present a hybrid approach that uses both materialization and query reformulation, which can be summarized as follows.

1. Generate a partial materialization of \(kb\) called schema closure, \(kb^+\), using simplified entailment rules Materialize\(^-\) that derives all non-rdf:type statements first. The computation of rdf:type triples is deferred until processing queries using reformulation (Materialize\(^-\)(kb) = kb\(^+\)).
2. Reformulate the query \(q\) using rules Reformulate\(^-\) that compute all statements derivable from rdf:type triples, producing \(q^{ref-}(Reformulate\(^-\)(q, kb\(^+\)) = q^{ref-})\)
3. Evaluate the reformulated query \((q^{ref-})\) over \(kb^+\), which produces complete answers \((q^{ref-}(kb^+) = q(kb^∞)\).

We also list the reformulation rules for Reformulate\(^-\) below because some of them are referred in
the next section.

\[
\langle ?s \tau ?o \rangle \in q \\
\langle ?x \subseteq sc ?o \rangle, \langle ?s \tau ?x \rangle \quad \text{R3} 
\]

\[
\langle ?s \tau ?e \rangle \in q \\
\langle ?c \subseteq sc ?e \rangle, \langle ?p \subseteq sp ?r \rangle, \langle ?r \subseteq d ?c \rangle, \langle ?s ?p ?x \rangle \quad \text{R4} 
\]

\[
\langle ?s \tau ?e \rangle \in q \\
\langle ?c \subseteq sc ?e \rangle, \langle ?p \subseteq sp ?r \rangle, \langle ?r \subseteq r ?c \rangle, \langle ?x ?p ?s \rangle \quad \text{R5} 
\]

Note that we omit rule R1 and R2 because these rules essentially generate statements that some subjects/objects are simply the type of class resources, which may not be useful as answers. R3 essentially captures all rdfs:subClassOf hierarchies. R4 and R5 handle instances represented using rdfs:domain and rdfs:range axioms with the combinations of rdfs:subClassOf and rdfs:subProperty.

2.3 SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL\(^3\) is a standard query language designed for data modeled using RDF. The main construct of SPARQL is a graph pattern which is a composition of triple patterns. A triple pattern is a triple with at least one variable (denoted by leading ’?’) in either the Subject, Property or Object positions. For example, the triple pattern (\(?\text{prs} : \text{name} ?\text{n1}\)) has two variables ?\text{prs} and ?\text{n1}. The semantics of a triple pattern is the subset of a triple model that is produced by substituting query variables in the triple pattern. For example, the common (join) variables ?\text{prs} and ?\text{crs} can be substituted as :Prs1 and :Crs1 based on triples in Fig. 1.1(a). Fig. 4.1(a) shows an example SPARQL query to retrieve a list of faculty and the courses that they teach. The SPARQL query also can be represented as a form of graph. Fig. 4.1(b) shows the graph version of the query Q. The answer to our example query is a 2-ary relation containing valid bindings for the variables ?\text{prs} and ?\text{crs} (here our result relation consists of a single tuple ⟨ :Prs1, :Crs1 ⟩).

2.4 The MapReduce Framework

2.4.1 MapReduce

MapReduce [31] is a parallel programming model that allows programs to be automatically parallelized over arbitrary-sized clusters of commodity-grade machines. Many different types of MapReduce im-

\(^3\)http://www.w3.org/TR/rdf-sparql-query
plementations are available and the most widespread one is Apache Hadoop\(^4\). Recent Hadoop releases support MapReduce as one of its distributed applications. Users only need to translate their functionality into a set of MR jobs which consist of the two primitive functions to implement a task using the MapReduce (MR) model, i.e., map and reduce. The encoded Map and Reduce functions are executed across multiple slave nodes in a Hadoop cluster where each node has a number of Mapper /Reducer slots to execute Map /Reduce functions independently.

Fig. 2.2 shows an overall execution flow of a map and reduce function using two physical slave nodes (\(DN_1\) and \(DN_2\)). The map function is defined as \(\text{map} \left(\text{Key1}, \text{Value1}\right) \rightarrow \text{list(\text{Key2}, \text{Value2})}\). Each node first reads input splits from Hadoop Distributed File System (HDFS), processes the input key-value pairs based on user’s codes described in the map function (\(M_1\) and \(M_2\)), and produces intermediate data as a list of key-value pairs. The reduce function is defined as \(\text{reduce} \left(\text{Key2, list(\text{Value2})}\right) \rightarrow \left(\text{Key3, Value3}\right)\). Each reduce node pulls intermediate data from nodes that execute Map functions, merges values into a group according to the intermediate key, executes user’s codes in reduce function (\(R_1\) and \(R_2\)), and finally generates a final key-value pair for each group. Each MR job is accompanied by I/O costs to materialize the outputs of map and reduce functions, sorting costs for grouping tuples with same keys and communication costs to move data between the Mappers and Reducers (nodes which execute the map and reduce functions respectively).

### 2.4.2 Hadoop Distributed File System (HDFS)

The physical unit of input and output in MR jobs is a file, which is loaded from and stored to HDFS (Hadoop Distributed File System). HDFS stores application data and filesystem metadata in a separate manner; data is stored across slave nodes called Datanodes, and metadata of logical files are stored in

\(^4\)http://hadoop.apache.org
a dedicated node called *Namenode*. Namenode also maintains in-memory mappings between logical files and physical data blocks in datanodes. Each data block consists of unstructured byte streams stored in physical files managed by a local file system of datanodes.

While HDFS is suitable for processing a small number of large files, the performance of HDFS quickly begins to degrade when a MR job need to process many small files, i.e. the small-files problem [18]. This is because the namenode becomes a bottleneck due to significant amount of network I/Os with datanodes, resulting in performance degradation or failed execution of MR jobs. It is also likely that a namenode experiences memory shortage due to large number of in-memory file mappings. Recently, Hadoop supports multiple namenodes using HDFS Federation [9], but the issue of handling many files still results in performance degradation.

To relieve the small-files problem, Hadoop provides a number of custom file formats that can be used to group files such as *SequenceFile* and *MapFile* [8]. *SequenceFile* is a flat file that stores multiple entries using a key-value pair format but does not support random access. *MapFile* provides key-based random access using two *SequenceFiles* (index and data). An index file contains keys and offsets of entries in the data file. When accessing a *MapFile*, its index file is loaded into a node’s memory, which allows for in-memory probing and random-access of entries in the data file based on keys.

### 2.5 SPARQL Query Processing on MapReduce

#### 2.5.1 Graph Pattern Matching Using Relational Algebra

RDF triples are commonly stored as ternary relations. In this context, SPARQL graph pattern queries can be interpreted as a set of scans and joins. For example, assuming that we partition the relation based on properties using the vertical partitioning [19] in advance, the query Q can be expressed using relational algebra as shown in Fig. 4.1(b). However, this interpretation often requires a significant number of joins to be processed. For example, the number of required joins is 7 in Fig. 4.1(b) even though the query is relatively simple. In reality, several join operations are typical in SPARQL graph pattern matching [21]. Therefore, SPARQL query processing often requires a significant amount of computing resources and processing time.

In the context of MapReduce, an interpretation of graph pattern queries often results in multi-cycle MR workflows because each join is generally mapped into a single MR cycle (or job) [24] (assuming that any particular optimization techniques are not employed). When tasks require more than one MR cycle (more often than not), the multiple iterations of I/O, communication and sorting costs can drive up the cost of processing. Consequently, proposing techniques to reduce the number of MapReduce cycles is crucial for optimization of SPARQL query processing performance.
2.5.2 Nested Triplegroup Algebra and Data Model (NTGA)

One optimization technique that may be used to mitigate the cost of joins on MapReduce is to aggressively group operations into as few cycles as possible. In this context, one observation that can be exploited is that i) graph patterns often consist of multiple star structures as sub-patterns and ii) all star subgraphs matching any star sub-patterns can be computed using a single GROUP BY operation. Based on this observation, an alternative algebra called the Nested TripleGroup Algebra (NTGA) was proposed in our previous works [65]. NTGA allows for processing RDF data in a more natural way – in terms of groups of triples sharing Subject/Object or triplegroups rather than a set of n-tuples in relational algebra.

For example, $tg_1$ in Fig. 2.3(a) is a triplegroup sharing a common subject $prs1$. These triplegroups can be computed using a single GROUP BY operation, thus NTGA does not require multiple MR cycles (one for each star subpattern) to evaluate multiple star subpatterns using relational-style joins.

Two advantages of the triplegroup model are (i) conciseness of representation, similar to the advantage of the nested relational model over the classic relational model and (ii) efficiency of computation. As an example of (i), the triplegroup $tg_5$ in Fig. 2.3(b) represents the same information as, or is content-equivalent to the set of n-tuples \{tup1, tup2\}. Intuitively, we can produce an equivalent set of n-tuples from a triplegroup by splitting the triples in the triplegroup based on the property types, and then applying a Cartesian product between the resulting subsets, e.g., $tg.triples(\text{foaf:name}) \bowtie tg.triples(\text{:emailAddr})$ where $tg.triples(prop)$ is the set of triples in $tg$ with property type $prop$. More precisely, since triples in a triplegroup all share the same subject, the content-equivalence relationship is always with the set of n-tuples resulting from a star-join. With respect to (ii), it is possible to compute all triplegroups represented in data (irrespective of structure) using an operation similar to a relational GROUP BY operation on the Subject column.

NTGA also provides a set of operators that create and manipulate triplegroups. Triplegroup-based processing model centers around the constituent star subpatterns in a graph pattern. At a high level, processing amounts to finding triplegroups that “match” star patterns and then “joining” these star subgraphs and intermediate results iteratively until the complete graph structure is constructed. We summa-

Figure 2.3: (a) a set of triplegroups corresponding to example triples and (b) content-equivalence between a triplegroup $tg_5$ and n-tuples $tup_1$ and $tup_2$. 

- $tg_1 = \{\{a, \{u2, u3\}\}, \{b, \{"a.com"\}\}, \{t, \{\text{Person, Faculty}\}\}\}
- $tg_3 = \{\{a, \{u2, u4, u6\}\}, \{m, \{"f@b.com"\}\}, \{n, \{"b"\}\}, \{t, \{\text{Person, Faculty}\}\}\}
- $tg_5 = \{\{h, \{"crs1.edu"\}\}, \{m, \{"crs1@n.edu"\}\}, \{n, \{"crs1\}\}, \{r, \{10\}\}, \{t, \{\text{Course}\}\}\}
- $tup_1 = \{\{\text{Stud3}, \text{foaf:name}, "Bill"\}, \{\text{emailAddr}, "c@n.edu"\}\}
- $tup_2 = \{\{\text{Stud3}, \text{foaf:name}, "Bill"\}, \{\text{emailAddr}, "c@g.com"\}\}

\[\equiv \text{(content-equivalent)}\]

\[tg_5 = \{\{h, \{"crs1.edu"\}\}, \{m, \{"crs1@n.edu"\}\}, \{n, \{"crs1\}\}, \{r, \{10\}\}, \{t, \{\text{Course}\}\}\}\]

\[\cong \{\text{Stud3}, \{\text{foaf:name}, "Bill"\}, \{\text{emailAddr}, "c@g.com"\}\}\]
Figure 2.4: (a) A relational query execution plan and (b) a NTGA-based query execution plan for query Q.

1. **TG_GroupBy (∧)**: Similar to the relational **GROUP BY** operator, this operator constructs triple-groups from triples sharing a common Subject.

2. **TG_GroupFilter (σ)**: This operator enforces structural constraints specified in sub star patterns over triplegroups resulting from **TG_GroupBy**. For example, **TG_GroupFilter** operator in Fig. 2.4(a) filters out **tg**₂ and **tg**₃ since they do not match structural constraints of star patterns. In other words, these triplegroups do not contain ‘all’ properties in star patterns, i.e., `{rdf:type, :affWith, :name, lectures}` or `{rdf:type, :hpage, :mbox, :name}`.

3. **TG_Join (⋈)**: A “join” operator defined on triplegroups that produces nested triplegroups as a result. Nested triplegroups have a root triplegroup and one or more child triplegroups. For example, the nested triplegroup **ntg** is produced by the **TG_Join** of **tg**₁ with **tg**₄ on the object of property :lectures in **tg**₁ with the subject of **tg**₄.

Fig. 2.4(b) shows the translation of the query Q in Fig. 4.1(a) to a MapReduce workflow which is executed in an extended Pig system that supports NTGA. In the map phase of the first cycle, **TG_Filter** selects triples that match triple patterns in a query, i.e. equivalent to the relational **SELECT** operator. **TG_GroupBy** groups selected triples and produces triplegroups. **TG_GroupFilter** then filters out triplegroups that fail structural constraints. The other cycles execute the **TG_Join** operations to join triplegroups. Note that the NTGA-based query execution plan in Fig. 2.4(a) uses only two MR jobs while the relational query execution plan in Fig. 2.4(b) requires three MR jobs. Generally, given a query with **n** star sub patterns, the MapReduce workflow generated from an NTGA-based plan consists of **n** jobs vs. **2n − 1** jobs for the relational approach.
Chapter 3

Optimizing the Processing of Queries with Semantically Related Subexpressions

3.1 Motivation

Many real-world semantic networks represented using RDF often consist of entities described using common vocabularies to ensure interoperability and support automatic interconnections with other networks. For example, most protein entities in Uniprot are often additionally described using predefined RDFS vocabularies such as rdfs:seeAlso, which is often used to provide additional information about entities. As a result, complex graph patterns often contain semantically related subexpressions, i.e., common subexpressions which consist of repeated properties over multiple star patterns. Fig. 3.1(a) shows a two-star-pattern query \( RQ \) that retrieves a list of products and their producers from datasets modeled based on e-commerce usecase scenario. Note that a set of properties are repeated across two star patterns \( \text{stp}_1 \) and \( \text{stp}_2 \) to retrieve labels and (registered) dates of resources typed as bsbm:Product and bsbm:Producer, e.g., rdfs:label (blue), dc:date (violet), and rdf:type (red).

Existing approaches often represent these subexpressions based on the first-order relational expressions and evaluate them independently, i.e., scan the input relation for that property as many times as it participates in join subqueries. Fig. 3.1(b) shows an execution plan for the query \( RQ \) (Properties/Objects are abbreviated due to space restrictions, e.g., 1 denotes rdfs:label). The plan requires scanning the input relations containing the property rdfs:label twice, once for each join in which the property participates. Similarly, relations containing rdf:type and dc:date are each scanned twice. Depending on the approach used to partition the joins into MR cycles, such repeated properties may result in redundant scans that increase the overall I/O overhead of such workflows. Hence, there is a need for approaches that enable scan-sharing while processing graph pattern queries with repeated properties.

To avoid such multiple scans on a relation, we may either buffer the relation for the duration it is needed (if memory is available) or we may use DAG (Directed Acyclic Graph) shaped plans so that
the output of an operator can be sent to more than one operator. This requires either inter-operator or pipelined parallelism which allows concurrent execution of operators to be enabled. However, neither one of these scenarios is possible in the MapReduce model. One of promising approaches to solve these issues is to reformulate the expression into semantically equivalent second-order expressions using NTGA. NTGA allows for evaluating all star subgraph patterns concurrently using a single TG-GroupFilter, thus it is also possible to concurrently evaluate all related subexpressions in the patterns. However, this model introduces challenges in query executions. In particular, an interpretation of semantically related subexpressions using NTGA needs to handle ambiguities. Because multiple star graph patterns include common subexpressions which need to be evaluated concurrently, a triplegroup can potentially match multiple star patterns while triplegroups are assumed to be mapped to a single star pattern in NTGA. Therefore, we need to extend our approach for managing and identifying triplegroups in a way that relaxes the assumption of a single star pattern for each triplegroup.

3.1.1 Contributions
In this chapter, we address the limitation by extending NTGA to identify and manage ambiguities. Specifically, we propose
1. A formal conceptual model that enables a shared evaluation of semantically related subexpressions.
2. A design and algorithm of Triplegroup Cloning for handling ambiguities in NTGA.
3. A comprehensive evaluation demonstrating performance benefits of cloning operations.

3.1.2 Related Work
There is a substantial body of research on exploiting common subexpressions for optimizing queries in conventional database systems. In particular, it has been extensively studied in the context of optimizing multiple related queries, i.e., multi-query optimizations (MQO) in [68, 87]. MQO is also studied in the context of optimizing multiple graph pattern matching queries [53]. In addition, there are also a number of studies that identify similar workflows and share the execution on MapReduce. For example, MRShare [62] identifies different types of sharing opportunities by analyzing MR job workflows.
generated from a set of queries and shares the execution or the output of Map/Reduce phases to reduce the cost of execution. However, these studies mostly focus on inter-query aspects that exploit common expressions across different queries for optimizations rather than intra-query ones that leverage related subexpressions in subqueries.

Recently intra-query optimizations have been explored in several studies. For example, Cascade-style optimizers in Scope [75] is extended to employ such optimization and select globally optimal plans based on its cost model. YSmart [55] leverages correlations between subqueries to reduce the cost of query execution on MapReduce. It employs a correlation-aware SQL-to-MapReduce translator that identifies correlated subplans and merges them based on a set of rewriting rules to minimize the number of MR jobs. While these works have some resemblances to the approaches introduced in this chapter, they often only focus on query transformation for optimization without transforming intermediate data to share the evaluation of common operations.

Some RDF query processing systems also employ intra-query optimization techniques such as Sideways Information Passing (SIP) in RDF-3x [61]. However, the goal of optimizations is often different, e.g., SIP focuses on reducing the cost of joins by exchanging meta information between join operators and removing irrelevant intermediate results in advance. However, it does not specifically eliminate the redundant evaluation of related subexpressions. These techniques are complementary to our approach, which could further improve the performance of graph pattern matching queries.

3.2 Foundations

3.2.1 Well-formed, Ambiguous, and Perfect Triplegroups

In the following description of basic concepts, we will assume the existence of an RDF dataset $D$ that is a collection of statements about properties $P_1, P_2, ..., P_n$. $D$ is represented as a set of vertically partitioned relations $P_1, P_2, ..., P_n$. We further assume the existence of the following convenience functions for any set $S$ of triples or triple patterns: $\text{props}()$ returns the set of distinct property names in the set; $\text{triples(propname)}$ returns the subset of $S$ whose properties are propname.

Definition 3.2.1. WELL-FORMED, AMBIGUOUS AND PERFECT TRIPLEGROUPS.

I. Let $Q$ be a query consisting of star patterns $stp_1, stp_2, ..., stp_k$ and $tg = \{t_1, t_2, ..., t_m\}$ be a triplegroup. $tg$ is said to be well-formed with respect to $Q$ if there exists a subset $tg'$ of $tg$ such that $tg'.\text{props}() = stp_i.\text{props}()$ for some $i$. In other words, a well-formed triplegroup contains triples whose properties overlap with some star pattern in a query and therefore can be considered relevant to the query. For example, the triplegroups $tg_0, tg_1$, and $tg_2$ in Fig. 3.2 are well-formed.

II. In addition to I., if there exists a subset $tg''$ of $tg$, which is not necessarily required to be disjoint with $tg'$, and $tg''.\text{props}() = stp_j.\text{props}()$ for any $j \neq i$, we say that $tg$ is ambiguous. Essentially, an
ambiguous triplegroup is one whose properties span multiple star patterns in a query; therefore, \( tg_0 \) in Fig. 3.2 is an ambiguous triplegroup.

III. Finally, we say that a triplegroup is perfect if \( tg.props() = stp_k.props() \) for exactly one \( k \). It is unambiguous and contains no properties except those in \( stp_k \). Perfect triplegroups are valid answers for a query. The triplegroups \( tg_1 \) and \( tg_2 \) in Fig. 3.2 are examples of perfect triplegroup.

![Figure 3.2: Cloning in the context of ambiguous triplegroups](image)

### 3.2.2 Triplegroup Cloning

Since the current TG_GroupFilter semantics assume perfect triplegroups as input, we need to extend its definition to deal with ambiguous triplegroups. Specifically, our approach is to have TG_GroupFilter split any ambiguous triplegroup \( ta \) into a set of subgroups (not necessarily disjoint) representing all perfect triplegroups that are derivable \( ta \) with respect to the query. However, care must be taken not to introduce redundant or invalid triplegroups which could result in spurious results. This is handled by a concept we call cloning.

**Definition 3.2.2.** TRIPLEGROUP CLONING Given a query \( Q \) consisting of star patterns \( stp_1, stp_2, ..., stp_n \) and an ambiguous triplegroup \( ta \), cloning creates a triplegroup \( ta' \subseteq ta \) if \( ta' \) is perfect with respect to some star pattern \( stp_i \).

Fig. 3.2 shows how perfect triplegroups \( tg_1 \) and \( tg_2 \) are cloned from the ambiguous triplegroup \( tg_0 \) and the non-perfect triplegroup \( tg_3 \) is rejected.

**Lemma 3.2.1.** Triplegroup cloning is lossless.

**Proof.** To prove losslessness, we need to show that no valid intermediate results are destroyed nor are spurious results introduced by cloning.
1. Since ambiguous triplegroups are not perfect for any single star pattern, they do not represent valid intermediate results for any star pattern. Consequently, decomposing an ambiguous triplegroup into subgroups does not destroy any valid intermediate subquery results.

2. Since cloning is applied only to ambiguous triplegroups, we do not introduce redundant triplegroups (redundant triplegroups will amount to duplicate answers in a star pattern subquery result).

3. Since each cloned triplegroup is perfect for a star pattern i.e. is a valid unique result for a star pattern subquery, the only other possible way of introducing redundancies is via the redundancies that exists within subgroups of different triplegroup clones (recall that the triples whose properties are repeated across star patterns will be repeated across the corresponding clones). However, each repeated property plays a different role in the query and separate bindings are needed for each role. This is analogous to have multiple copies of a relation for a self join. When these the different properties are eventually joined, there will be a single occurrence of the property for each of its roles which is equivalent to the result produced using the relational join operator.

Since cloning is lossless, the final results obtained from joining such intermediate triplegroups using the \texttt{TG.Join} operator are correct and complete. This is straightforward due to the above lemma and the correctness of our already existing \texttt{TG.Join} operator which joins intermediate triplegroup results. Algorithm 1 shows the extensions to \texttt{StarGroupFilter} to support handling of ambiguous triplegroups. In the map phase, the tuples are annotated based on the $S$(subject) component (line 1). In the reduce phase, all tuples corresponding to the same subject component are processed in the same reduce function. A bitmap (\texttt{locBitset}) is used to keep track of the property types processed (line 3) and the (Property, Object) pairs are stored in a temporary map (\texttt{tempMap} in line 4). After processing all tuples in the group, the locBitSet is matched with all the equivalence classes (star subpatterns) in the query (\texttt{ECList} in line 5). A match with a single equivalence class represents a perfect triplegroup (line 10). A match with more than one equivalence class indicates an ambiguous triplegroup and the temporary map is cloned to retrieve the relevant (Property, Object) pairs corresponding to the matched star subpatterns (lines 7-8). The cloned maps are then used to create perfect triplegroups (line 9) corresponding to the star subpatterns required in the query. A mismatch with all of the star subpatterns results in filtering out of the group of tuples.

### 3.3 Empirical Evaluation

The goal of our evaluation is to compare the performance of the relational style and NTGA based approaches and their ability to share scans while processing query patterns involving repeated properties ($DupPs$). Three approaches were considered to evaluate the redundant scans caused by $DupPs$ during star-join computations, (i) 1-join-per-cycle (SHARD), (ii) 1-star-join-per-cycle (VP approach described
Algorithm 1: StarGroupFilter

1 Map (key:null, val: Tuple tup\langle s,p,o \rangle);  
2 emit \langle s,tup \rangle;  
3 Reduce (key:Sub, val:List of tuples T);  
4 foreach tup\langle s,p,o \rangle \in T do  
5 \hspace{1cm} set p in locBitset;  
6 \hspace{1cm} add \langle p,o \rangle to tempMap;  
7 matchedList = match(locBitSet, ECList);  
8 if \langle \text{matchedList} \rangle > 1 then  
9 \hspace{1cm} //Ambiguous TripleGroup  
10 \hspace{1cm} foreach EC \in \text{matchedList} do  
11 \hspace{1cm} \hspace{1cm} propMap \leftarrow \text{cloneMap}(tempMap, EC.propList);  
12 \hspace{1cm} \hspace{1cm} emit \langle RDFMap(Sub,EC,propMap) \rangle;  
13 \hspace{1cm} else  
14 \hspace{1cm} \hspace{1cm} //Perfect TripleGroup  
15 \hspace{1cm} \hspace{1cm} emit \langle RDFMap(Sub,\text{matchedList}[0],tempMap) \rangle;

in section II.B.1), and (iii) all-star-joins-1-cycle (NTGA approach). Four tasks are designed for the evaluation by varying the selectivity of DupPs (Task 1), the number of DupPs (Task 2), the ambiguity of triplegroups (Task 3), and the size of graphs (Task 4).

3.3.1 Testbed Setup

Setup The experiments were conducted on 10-node clusters in VCL\(^1\), an on-demand virtual computing environment. Each node was equipped with a dual core intel x86 processor (2.33 GHz) and 4GB memory. Pig 0.8.0 and Hadoop 0.20.1 were installed and configured with 256MB for block size and 1GB for the heap-size of child JVMs. We chose Apache Pig that supports relational-style operators (Pig-Def), and extended it to include the NTGA operators (NTGA). Jena\(^2\)'s ARQ was integrated into Apache Pig to interpret SPARQL queries. SHARD[66] uses a pre-processing MR job to transform all the star-subgraphs (edges sharing the same subject node) into a single record, e.g., triples of a triple-group \(tg_1\) in Fig. 3.2 are represented as “s1, (homepage, o1, country, o2)” in SHARD. Other additional MR jobs such as to process the SELECT clause in a SPARQL query, were disabled. All results recorded were averaged over two or more trials.

Testbed The BSBM benchmark tool\(^3\) was used to generate four synthetic datasets using number of Products as the scalability factor – BSBM-{250k, 500k, 750k, 1000k}, with the size of data ranging from 22GB (BSBM-250k with 250k Products and approx. 86M triples in total) to a data size of 87 GB (BSBM-1000k with 1M Products and approx. 350M triples in total). BSBM dataset includes 7 repeated properties - some defined across all classes e.g. type, publisher, and some defined for a smaller subset of classes, e.g. label. The size and selectivity\(^4\) of each DupP in BSBM-250k is as follows: publisher

\(^1\)http://vcl.ncsu.edu
\(^2\)http://jena.sourceforge.net
\(^3\)http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/
\(^4\)Selectivity(P) = \frac{\mid T_P \mid}{\mid T \mid} where T_P denotes triples containing P and T denotes all triples
{1.7GB, 0.091}, type \{1.8GB, 0.105\}, label \{49MB, 0.003\}, and date \{1.4GB, 0.091\}. Additional details are available on the project website \(^5\).

### 3.3.2 Evaluation Results

**Task1: varying the selectivity of DupP** The task evaluated two queries with varying selectivity of DupPs: (i) rq1 with low selectivity DupP (label) and (ii) rq2 with high selectivity DupP (publisher), both having two star subpatterns each. The experiments showed that NTGA has around 42% performance gain over Pig-Def for rq1 and increases to around 48% gain for rq2. With the higher selectivity DupP (rq2), Pig-Def uses additional 70 seconds than rq1 due to the overhead of redundant scans. It was observed that NTGA showed similar performance for both rq1 and rq2 and hence, performs better than Pig-Def for queries with varying selectivity of DupP.

![Figure 3.3](http://research.csc.ncsu.edu/coul/RAPID/cloud2012_exp.html)

**Task2: varying the number of DupPs** This task studies the effect of increasing the number of DupPs across a query with two star-joins for BSBM-250k using the three approaches. Four queries (dq0 to dq4) containing two star patterns were considered, with equal numbers of unique properties and varying number of DupPs (from 0 to 4 respectively) in the second star subpattern. The queries include the following DupPs: dq0 (none), dq1 (publisher), dq2(publisher, type), dq3(publisher, type, label), and dq4(publisher, type, label, date). Fig. 3.4(a)∼(c) shows the execution time and HDFS read/write for this task using the three approaches. In general, the 1-join-per-cycle approach in SHARD results in highest execution time and I/O compared to other two approaches. Further, the execution time in SHARD increases as the number of triple patterns increase from 8 to 12 in dq0 to dq4 respectively. For example, the query dq4 is processed using 13 MR cycles in SHARD (including MR1 for pre-processing), 4 MR cycles in Pig-Def and 2MR cycles using the NTGA approach as shown in Fig. 3.3. Note that the execution time of the MR1 in SHARD is longer than the grouping phase of NTGA because SHARD stores the output as raw text format without using any compression or format; such trends were

\(^5\)http://research.csc.ncsu.edu/coul/RAPID/cloud2012_exp.html
observed across other queries. Pig-Def shows relatively better performance compared with SHARD because the number of MR jobs for star-join is mainly affected by the number of star patterns in the queries. Additionally, the `SPLIT` operator in Pig-Def triggers the insertion of a preceding map-only job (Pig-Def’s MR1 in Fig. 3.3), which helps to reduce some HDFS read I/O in their subsequent jobs due to the compression of the output of MR1. However, the amount of HDFS reads for all queries in Pig-Def is still larger than in NTGA. This is because the DupPs are scanned and processed in BOTH the star-join cycles (MR2 and MR3) in Pig-Def, which results in an increasing amount of HDFS bytes read as the number of DupPs increase. However, in NTGA, it is observed that the execution time and the amount of HDFS reads does not change much with varying numbers of DupPs. This is due to the grouping-based star-join computation approach in NTGA that enables scan-sharing while processing star subpatterns containing DupPs. Additionally, NTGA shows relatively small amount of HDFS written compared with the other approaches due to the compact representation of intermediate data.

**Task3: varying the frequency of DupPs in a query** This task focuses on the evaluation of the three approaches with the increasing occurrence of a DupP across a graph pattern query. Four queries (mq1 ~ mq4) were considered, with varying number of star patterns (1 to 4). Two DupPs occur in each of the star subpatterns, thus increasing the frequency of DupPs from 2 (in mq1) to 8 (in mq4) respectively. The execution time and the amount of HDFS bytes read/written for this task are shown in Fig. 3.4(d)~(e). SHARD shows significantly more I/O and execution time than the other two approaches because the number of required MR jobs increases four times i.e. 4 (for mq1) to 16 (for mq4) respectively. In Pig-Def, continuous increments of HDFS reads is observed as well because the number of MR jobs required for star-join increases from 1 in mq1 to 4 in mq4. In NTGA, a clear benefit of scan-sharing is observed.
Though the occurrence of DupP across increasing number of star patterns leads to an increasing number of ambiguous triplegroups in NTGA, no significant overhead of the cloning process is observed. Similar to Task2, the HDFS writes using NTGA approach is relatively smaller than the other approaches.

**Task4: varying the size of graphs** Fig. 3.5 shows a comparative evaluation of the two approaches with increasing number of RDF triples for query dq4 used in Task2. The NTGA approach scales well with a performance gain of 52% to 58% ranging over BSBM-250k to BSBM-1000k data sizes, respectively.

### 3.4 Chapter Summary

In this paper, we extend our previous efforts on algebraic optimization of RDF graph pattern queries to enable efficient handling of graph pattern queries with multiple occurrences of a property type, which is a common scenario in RDF queries. Our approach formalizes and integrates the concept of ‘cloning’ as part of appropriate operators of our NTGA algebra, avoiding the need for multiple scans of input relations required in relational algebra-based query plans. Our extensive experimental evaluations with various workloads have shown the effectiveness and scalability of our intra-query scan-sharing approach.
Chapter 4

Automatic Semantic Typing Scheme for Semantics-oriented RDF Storage Model on Hadoop

4.1 Motivation

Type-based optimization [22, 59] has been used for tasks ranging from checking validity of queries to pruning state-space for optimizing queries in relational and semi-structured databases. It has also been foundational to some Semantic Query Optimization techniques [22, 58](SQO) where together with the use of integrity constraints, more complex query rewritings or reformulations into efficient queries are achieved. SQO refers to techniques that use integrity constraints to eliminate subexpressions from queries (often joins). But in the case of the growing amount of data represented using RDF, SQO has not been considered for a few reasons. For one thing, the RDF model does not have a strong typing model that is required for type-based optimization, nor does it have integrity constraints in the true sense.

Consider the RDF graph in Fig. 1.1(a), which represents knowledge in the academic domain. In storing and querying such RDF graph in a relational model, we could place all triples in a single relation or create separate relation for triples with Property type a (:affWith), one for m (:mbox), one for h (:homepage) and so on. However, this approach does not lend itself to a strong typing model where entities/resources have unique type assignments. For example, the Subject :prs2 would belong to multiple relation types. To further complicate matters, RDF data can be accompanied by an ontology which provides definitions of Classes and Relationships (Properties) that resources can explicitly assert membership about. For example, the triple (:Faculty, rdfs:Class, :Faculty) asserts that the identifier :Faculty is a class, while the triple (:prs1, rdf:type, :Faculty) asserts that the Subject :prs1 is the instance of the class :Faculty. However, the members of the same class, e.g., :Faculty, does not necessarily have the same Properties describing them. For example, :prs1 and :prs2 are both members
of class :Person, but they have a different list of Properties describing them, i.e., \{a, h, l, m, n, t\} and \{a, m, n, t\}, respectively. The traditional notion of types tends to partition data into equivalence classes where all instances of a type can either meet some constraint or not (and in the latter case may be pruned out entirely). Given the above discussion, it is unclear what would be the notion of ‘types’ that could induce appropriate partitioning and equivalence classes on resources in an RDF model.

However, imposing a node-oriented perspective on RDF may offer better typing possibilities. Fig 4.1(b) shows a partitioning of triples represented in the example knowledge graph into sets / groups based on their Subject equivalence, i.e., having the same Subject. (This corresponds to the notion of triplegroups in [65, 48]). Consider the left star structure of a graph pattern in Fig. 4.1(a) which seeks to match star subgraphs with Properties \{a, 1, n, t\} from the partitioning in Fig. 4.1(b). We observe that triplegroups in the partitioning that ‘match’ the star subgraph (i.e., triplegroups with Properties \{a, 1, n, t\} such as \(tg_1\)) also contain the Object :Faculty. In other words, we can consider the need to match Properties \{a, 1, n\} as well as Property and Object (\texttt{rdf:type} and :Faculty) as superfluous. In the spirit of SQO, we can eliminate that branch (denoted by dotted lines and ovals) in the graph pattern before executing the query. Given that the computation of each branch in the query is equivalent to a join operation, this is analogous to \textit{join elimination}.

Further, ontology languages such as RDFS provide axioms for the derivation of implicit facts in a model. By integrating the node-oriented perspective with implications of relevant ontological axioms, we arrive at some useful characterizations. For example, a predefined meta-Property \texttt{rdfs:domain} defines the domain of a Property as the set of classes that it can be applied to, e.g., the ontological axiom (\texttt{:affWith, rdfs:domain, :Person}) implies that resources that have the Property \texttt{:affWith} can be inferred to be of class :Person (even if that assertion is not explicitly made). With this axiom, the triple \((\texttt{prs1, :affWith, :u2})\) allows us to infer the triples such as \((\texttt{prs1, rdf:type, :Person})\). Therefore, it

![Figure 4.1: (a) Graph representation of graph pattern query Q that retrieves faculties who lecture courses (b) star subgraphs (triplegroups).](image-url)
may be possible to avoid having such ‘redundant’ triples explicitly represented in a data model. For example, we can dematerialize such inferable triples from $\tau_{g_1}$, $\tau_{g_2}$, and $\tau_{g_3}$ (denoted with check marks) in Fig. 4.1(b). This could reduce the size of relations and the cost of joins over such input relations. Such inferable triples often constitute a non-trivial proportion of triples in many real-world datasets, e.g., approx. 8% of DBPSB [60].

**Challenges.** The above discussion hints at the possibilities of (i) using some equivalence relationships for determining whether a join clause is trivially true and therefore can be eliminated from a query and (ii) manipulating the data itself (dematerializing triples) based on a notion of redundancy that is obtained when ontological axioms are combined with such equivalences. To avoid burdening users with knowledge about a new typing model and what types exist in a dataset, this typing model will need to be *autonomic* (self-managing). We argue that some notion of *typing* that builds on the equivalence relation in the earlier discussion could provide the right basis for reasoning about these issues. However, to preserve matching correctness with respect to SPARQL queries, triples cannot be arbitrarily dematerialized losslessly. Finally, when considering large scale processing in a distributed setting such as Apache Hadoop, some additional issues with respect to storage of typed data and distributed query processing over typed model arise.

### 4.1.1 Contributions

In this component, we make the following novel contributions:

1. **Autonomic Semantic Typing of RDF Data.** The *autonomic* typing scheme has three main dimensions - (i) the definition of a typing model for RDF (*R-Typing*) that is based on a very natural building block for RDF querying; (ii) a MapReduce-based algorithm for automatic and scalable induction of an R-Typed model from a triple-based RDF data model; (iii) *SemStorm* - a distributed storage system that introduces a polymorphic (*polytyped*), column-oriented file format for efficiently managing R-Typed models in HDFS.

2. **R-Type-based Query Execution Model for Hadoop** that introduces new logical and physical query operators for more efficient processing of SPARQL graph pattern queries over R-Typed models and a query rewriting process for translating SPARQL algebra queries to expressions over R-Types.

3. **Semantic Optimizations** that leverage ontological axioms similar to integrity constraints to (i) optimize storage models by identifying and losslessly dematerializing ‘redundancies’ in R-Type model - (triples that can be inferred by considering other triples in the type) and (ii) optimize query expressions by rewriting queries into more efficient ones given the relevant constraints. The overall effect of these optimizations is a reduction of a storage footprint and processing costs during query processing.

4. A comprehensive evaluation conducted on datasets of real-world (DBPSB) and synthetic (LUBM). The evaluation results show that our approach was up to 500X faster than existing ones.
4.1.2 Related Work

RDF data is often managed as a single ternary relation or a set of partitioned relations (very often vertically-partitioned relations where each Property type is a binary relation (Subject, Object) [19]). Fig. 4.1(b) shows an example graph pattern (variables are denoted with leading ‘?’). Fig. 4.1(a) shows an equivalent relational algebra expression for the graph, assuming a set of vertically-partitioned relations for each property, e.g., the :affWith relation. The algebraic expression illustrates the point about the number of joins needed, even for simple queries.

Most existing approaches [61, 41, 5, 33, 39, 43, 54, 19, 34, 44] employ fine-grained logical and physical data models with triples as first-class citizens. Such approaches are unable to avoid the need for large number of joins, but try to mitigate the cost of join operations using techniques such as multi-indexing schemes [61, 41] that enable more merge-joins, co-partitioning schemes [34, 43, 54] based on different notions of relatedness, that co-locate related triples to force more localized join processing rather than distributed joins.

Recent techniques [65, 88, 86, 64, 28] have adopted more coarse-grained first-class citizens for their logical and/or physical data models, enabling subgraphs of ‘related’ triples to be managed holistically. These techniques enable reduction in the number of joins required for query processing (depending on how coarse the relatedness criteria is). While processing in RAPID+ [65] requires a full scan of all data units or triplegroups, [88, 86, 64, 28] use a notion of similarity to cluster similar subgraph structures. The rationale for this is that such subgraphs are likely to be frequently part of the same results and retrieval costs can be reduced by clustering. The storage models that have been employed include n-ary relations [64, 28] where the properties in the subgraph structures form the schema of the relation or native indexing mechanisms [88, 86] based on some encoding of these structures into signatures that can be used as keys for an index. Such indexed access greatly reduces the search space for matching subgraphs to a pattern that has been translated to the same signature encoding. Of the above-mentioned techniques, only [65, 86] focus on partitioning such coarse-grained objects in a cluster for distributed processing, in particular, Hadoop processing. EAGRE [86] proposes a block format for storing its data objects using space-filling curve to optimize queries with modifiers such as ORDER BY.

4.2 Type-Based Optimization of RDF Graph Pattern Queries

Our type optimization strategy begins with (i) a typing model for RDF (R-Typing) for triplegroups, (ii) a strategy for autonomic management of R-Typed models, i.e., autonomous R-Type model generation from a triple model as well as automatic translation of queries expressed in the traditional triple-based algebra into NTGA expressions over R-Types, and (iii) an R-Type storage organization for Hadoop that is based on a polymorphic, column-oriented file format for HDFS. An important focus in this chapter is on distributed storage and query processing for enabling large scale RDF processing. In this regard,
we will be presenting a solution that uses Hadoop as the distributed framework. However, we want to emphasize that the solution we present is not necessarily limited to Hadoop’s computational model.

### 4.2.1 R-Type: A Typing Model for RDF

The R-Type model begins with triplegroups as first-class objects and is defined in terms of the property type combinations present in triplegroups, i.e., given $P$ as the set of possible Property types, an R-Type model is a function of $2^P$. An R-Type is denoted as $\tau$ with an associated Property combination as subscript. For example, given $P = \{a, h, l, m, r, t\}$, $\tau_a$, $\tau_{ah}$, ..., $\tau_{ahlmnrt}$ are possible types. These R-Types can be assigned to triplegroups based on Property type combinations in triplegroups. Given a set of triples whose Subjects are $s_1$ and Property types are $\tau_{p_1}$, ..., $\tau_{p_i}$, the corresponding triplegroup that comprises the same set of triples is assigned the R-type $\tau_{p_1p_2...p_i}$. More formally, let $\Gamma$ be a typing environment that assigns a triple $\langle s_1, p_1, o_1 \rangle$ based on its Property type $\langle s_1, p_1, o_1 \rangle$:

$$\Gamma \vdash \langle s_1, p_1, o_1 \rangle : \tau_{p_1}, \langle s_1, p_2, o_2 \rangle : \tau_{p_2}, ..., \langle s_1, p_i, o_i \rangle : \tau_{p_i}$$

$$\Gamma' \vdash \{ \langle s_1, p_1, o_1 \rangle \} : \tau_{p_1p_2...p_i}$$

It is straightforward to verify that such a typing model defines an equivalence relation on the set of triplegroups, i.e., each triplegroup (and therefore each triple) is assigned exactly one type.

**Example 4.2.1.** (R-Typed Triplegroups) The following types can be induced from the triplegroup model in Fig. 4.1(b): $\{\tau_a$, ..., $\tau_{ah}$, ..., $\tau_{ahlmnrt}\}$. The types can be assigned to triplegroups as follows: $tg_{1}^{a}$,$\tau_{ahlmnt}$, $tg_{2}^{a}$,$\tau_{amnt}$, $tg_{3}^{a}$,$\tau_{amnt}$, and $tg_{4}^{a}$,$\tau_{hmnrt}$.

### 4.2.2 Autonomic R-Type Model Generation and Storage on Hadoop

To exploit R-Typing for query optimization, we must not only re-interpret graph pattern queries in terms of R-Types but also enable discriminatory storage access of only query-relevant types. In this section, we present an R-Type storage model for HDFS, *SemStorm*. Considering the architecture and characteristics of Hadoop, a couple of design options are possible for storing R-Types on HDFS.

- **Single-R-Type-Per-Logical-File.** Here, each R-Type is stored in a separate logical file (which is physically partitioned across the cluster). This approach allows discriminatory type-based retrieval since instances of the same R-Type are stored in a file. Also, partitioning of files allows parallel retrieval and processing of the different partitions is possible. However, in Big Data scenarios where heterogeneity in data leads to a significant number of R-Types, this will produce many HDFS logical files (small
files for infrequently occurring types). This creates the well-known bottleneck at the Namenode due to the large amount of metadata that it needs to manage.

- **Multiple-R-Types-Per-Logical-File.** To avoid performance degradation due to many small files, we can group multiple R-Types into a single logical file. This notion of having multiple-types per-file is referred to here as a polymorphic-typed file organization. Two grouping strategies can be considered for a distributed storage organization: Horizontally-Partitioned Grouping which stripes each R-Type presented in a logical file across its corresponding physical files distributed in the cluster. Vertically-Partitioned Grouping does not stripe R-Type data. Instead, all instances of an R-Type are completely contained within one of the physical files associated with a logical polymorphic-typed file. Here, multiple R-Types can be in a single physical file. The Vertically-Partitioned Grouping allows for discriminative retrieval based on R-Types, but centralizes I/O and processing of an R-Type to the cluster node that hosts its file. On the other hand, the Horizontally-Partitioned Grouping allows parallelism. However, some indexing strategies will be necessary for identifying different stripes of an R-Type in different physical files.

The sequel presents a file organization, the Polymorphic-Typed Triplegroup file format (or PTG file format) that implements the concept of a polymorphic-typed file with horizontally-partitioned grouping.

**Polymorphic-Typed TripleGroup (PTG) File Format on HDFS**

The PTG file format requirements include (i) efficient discriminatory retrieval of specific (query-relevant) R-Types from a polymorphic-typed file that stores multiple R-Types, (ii) the projection of query-relevant Property types from matching R-Types (given that many matching R-Types many contain extraneous property types as we will see later), and (iii) achieving (i) and (ii) while keeping the overall number physical files to a manageable number that avoids overburdening HDFS. Both requirements suggest an indexed file organization that allows isolation of specific R-Types as well as isolation of components or fields of an R-Type. To these ends, we propose a column-oriented file format for storage of multiple R-Types in a file. Although, some column-oriented file formats for HDFS such as ORC [42] and Parquet [10] exist, they do not support the polytype file organization where multiple types can be stored in the same file, each exhibiting different striping characteristics.

The PTG file format is based on a SequenceFile and has the following characteristics:

- Each k/v entry consists of a key (Property associated with a type) and a value (Subject/Objects associated with Properties in triplegroups). All meta information such as an offset/length and the size of each k/v entry is stored in a global index file (PTG Index, which will be discussed later). The use of a k/v format with a global index allows the flexibility needed to i) map many different types to a single file and ii) selectively retrieve triples in matching typed triplegroups.

- Our format stores Objects of multi-value Properties in a denormalized manner, e.g., multiple Objects for Property a in \(tg_1\) are stored as a list such as \{:\u1, :\u3, :\u5\} in HDFS block 7 of Fig. 4.2. This
reduces required storage compared to normalized tuples.

Fig. 4.2 shows how triplegroups in Fig. 4.1(b) can be stored across two datanodes $N_1$ and $N_2$. Each datanode includes three types of files: i) a data file based on PTG file format ($PTG_i$), ii) an index file that supports a selective retrieval of triplegroups based on types ($PTGI_i$), and iii) a statistic file that stores size information on types ($SZ_i$). In node $N_1$, $PTG_1$ contains two types $\tau_{amnt}$ and $\tau_{ahlmnt}$, which are spanned across HDFS block 7 and 8 (Other details on SequenceFile such as sync markers are not illustrated for brevity). First, $tg_1$ and other triplegroups having type $\tau_{amnt}$ are stored between offset 0 and 55. Note that all Subjects and Objects for that type are consecutively stored using column-oriented layout. For example, all Subjects are stored between offsets 0 and 11 and similarly all Objects having Property $a$ (e.g., $\{u1, u3, u5\}$) are stored between offsets 11 and 21.

The offset and length information of each column is stored in an inverted index called PTG Index ($PTGI$). Each index file is associated with a PTG file and implemented using a SequenceFile format. Each entry in PTG Index is a k/v pair:

$$< p, \{< \tau_{ps1}, (off_1, len_1) >, \ldots, < \tau_{psk}, (off_k, len_k) > \} >$$

where the key is Property $p$ and the value is a set of types $\tau_{ps1}, \ldots, \tau_{psk}$. Each type entry includes an offset and the size of a column ($off$ and $len$) in PTG files where the column represents Subjects or Objects associated with Property $p$ in triplegroups. Based on these offset/length information, relevant triples in matching typed triplegroups can be selectively retrieved. Fig. 4.2 shows that the PTG index file $PTGI_1$ in HDFS block 1 contains a set of k/v pairs, e.g., Property $a$ is associated with a set of types that contains Property $a$, i.e., $\tau_{amnt}$ and $\tau_{ahlmnt}$.

Our initial approach to indexing offset and size of typed triplegroups based on Objects resulted in large number of accesses to HDFS (one per matching type). This access pattern resulted in degraded query performance because of excessive communication between Namenodes and Datanodes in addition to local disk accesses. Thus, we focused on minimizing HDFS access by augmenting position information at the end of a corresponding column entry in PTG indexes. Fig. 4.2 shows that (11, 10) is the offset and length of $\tau_{amnt}$, which is augmented to its R-Type entry $\tau_{amnt}$ associated with Property $a$ in $PTGI_1$.

### 4.2.3 Implementation of R-Type Model Generation

The remaining issue is how to autonomically induce R-Types from RDF data model and build R-Type storage model (SemStorm) on Hadoop. To address this issue, we introduce our implementation strategy for R-Type model generation, which consists of 3 steps: i) construct triplegroups, ii) dynamically partition them based on their R-Type and iii) flush partitioned triplegroups into corresponding entries of PTG files and indexes. We implement the strategy using a single MR job in the context of a system...
Figure 4.2: A physical representation of typed triplegroups using polymorphic-typed file organization and file format.

called RAPID+ [65] which is an extension of Pig\(^1\) that includes optimizations for RDF data processing. Fig. 4.3 shows the execution flow of those functions. The Pre-process plan generator in RAPID+ builds a plan that first constructs triplegroups by grouping triples by their Subjects. A Map() reads each triple and tags it based on its Subject. Subsequently, the operator TG Package in Reduce() constructs triplegroups from triples sharing the same Subject. The Reduce() can be summarized as follows.

\textbf{Algorithm 2:} Reduce phase - TG\_Package

\begin{verbatim}
// Constructing triplegroups from triples.
Reduce (k:subj, v:list of triples $t_1, t_2, ..., t_k$ having a common subj)
props = list of Properties $p_1, p_2, ..., p_k$ in $v$;
objs = list of Objects $o_1, o_2, ..., o_k$ in $v$;
triplegroup $tg = (subj, \{ (k:props, v:objs) \})$;
emit $tg$;
\end{verbatim}

The latter part of the Reduce() is handled by the PTG File Writer component that types triple-
groups automatically and serializes them into a PTG file. The partitioning step uses a buffer per type (Type Buffer), which is created on demand. This buffer is implemented based on FileBackedOutputStream class in Guava\(^2\). Each type buffer accumulates triplegroup instances in memory as much as possible and spills to disk if a threshold is reached. The threshold buffer size can vary based on the amount of memory available in physical nodes. The algorithm that partitions triplegroups is shown in Algo. 3. The PTG File Writer also sends metadata on typed triplegroups such as their types, offsets and lengths to the next component in the workflow, the PTG/SZ Index Writer.

**Algorithm 3:** Post-reduce phase - PTG File Writer

1. **PTGFileWriter** (triplegroup \(tg\) whose type is \(\tau_{ps}\))
2. check a type buffer (\(tgb\)) exists for \(\tau_{ps}\);
3. if \(tgb\) does not exist then
4. \(tgb = \text{new type buffer for } \tau_{ps}\);
5. store \(tg\) (without Properties) in \(tgb\);

Based on given metadata, the PTG/SZ Index Writer component constructs and writes PTG Indexes and SZ Indexes on HDFS.

### 4.3 Query Processing on SemStorm

#### 4.3.1 Query Compilation

To support query execution, SPARQL queries need to be compiled into query expressions over R-Typed models. We begin by introducing some notations and convenience functions for the formalization of the query processing model. \(\text{sig()}\) is a type signature function that returns the signature or property types that define a type, e.g., \(\text{sig}(\tau_{almnrt}) = \{a,l,m,n,r,t\}\). \(\text{props()}\) is a function such that \(\text{props}(tg)\) for a

\(^2\)https://github.com/google/guava
triplegroup \( tg \), returns the distinct set of property types in \( tg \), e.g., \( \text{props}(tg_2) = \{a, m, n, t\} \). \([\_]\) is an interpretation function such that \([\tau_i]\) returns all member triplexes for \( \tau_i \), i.e., triplexes \( tg \) such that \( \text{props}(tg) = \text{sig}(\tau_i) \). For example, \([\tau_{ahlmnt}] = \{tg_1\} \) for triplexes in Fig. 4.1(b).

To formalize the semantics of graph pattern queries under an R-Type model, we adopt the view that graph patterns can be seen as composed of star patterns linked by combination operations such as JOIN, UNION, OPTIONAL, etc. Given a graph pattern \( GP \) that comprises the set of star patterns \( stps = \{stp_1, stp_2, \ldots, stp_n\} \), \( GP \) can be expressed as a 2-tuple

\[
\{(stp_1, stp_2, \ldots, stp_m), \{op_1, op_2, \ldots, op_n\}\}, m \geq n
\]

where \( op_k \ (k \in 1 \ldots n) \) is an NTGA binary operator such as \( \text{\textbf{?}} \) with a pair \( stp_i, stp_j \) as operands. For example, a query \( Q \) in Fig. 4.1(a) can be decomposed into \( stp_{\text{ahlnt}} \) and \( stp_{\text{hmnt}} \) connected via a common variable \( \text{?crs} \). Consequently, given an R-Type model \( T \) that consists of a set of types \( T = \{\tau_1, \tau_2, \ldots, \tau_i\} \), we define the semantics of graph pattern queries under \( T \) inductively, beginning with the semantics of star graph patterns.

**Definition 4.3.1.** (Star Pattern Interpretation Under R-Typing) Given a star pattern \( stp \), the semantics of \( stp \) or \([stp] \), is defined in terms of two query operators:

i) **TG_Projection** \( (\pi) \): Given \( T \) and a set of properties \( p_1, p_2, \ldots, p_k \in P, \pi_{\tau_1,\tau_2,\ldots,\tau_k}(T) \) returns the projection of \( \{p_1, p_2, \ldots, p_k\} \) on \( T \), defined as a set of triplexes:

\[
\{tg' \mid tg' \subseteq tg, tg \in [[\tau_i \in T]], \text{props}(tg') = \{p_1, p_2, \ldots, p_k\}\}
\]

**TG_Projection** eliminates triplexes with property types that are irrelevant to a star graph pattern. For example, \( \pi_{\text{sig(stPattern)}}(T) \) returns \( \{tg'_1, tg'_2\} \) which correspond to \( tg_1 \) and \( tg_4 \) without irrelevant triples such as \( (\text{prs}1, a, n2), (\text{crs}1, r, "10") \).

ii) **TG_TypeSelection** \( (\sigma) \): Given \( T \) and a set of properties \( p_1, p_2, \ldots, p_k \in P, \sigma_{\tau_1,\tau_2,\ldots,\tau_k}(T) \) returns the following set of R-Types:

\[
\{\tau_1, \tau_2, \ldots, \tau_i\} \text{ where } i \supseteq \{p_1, p_2, \ldots, p_k\}
\]

Assuming that we overload the function \( \text{sig()} \) to produce the set of property types in a star pattern, **TG_TypeSelection** computes a set of R-Types that matches a star pattern \( stp \), i.e., a set of R-types that contains the star pattern’s property types as a subset of their signatures. For example, \( \sigma_{\text{sig(stPattern)}}(T) \) returns a set of types \( \{\tau_{ahlmnt}, \tau_{hmnt}\} \).

\([stp]\) can be expressed in terms of the above operators as follows.

\[
[[stp]] = \pi_{\text{sig(stp)}}(\sigma_{\text{sig(stp)}}(T)) \quad (4.1)
\]

35
In other words, we first apply a meta-subquery that produces a set of matching R-Types, from which we project relevant subgroups from type-matching triplegroups.

For the interpretation of a graph pattern that consists of multiple star patterns, we do not provide a formal definition due to space constraints. However, we include an example as follows.

**Example 4.3.1.** (Graph Pattern Interpretation) A query \( Q \) in Fig. 4.1(a) consisting of star patterns \( stp_{alnt} \) and \( stp_{hmnt} \) connected via a common variable \( ?crs \) can be expressed as:

\[
\left( \pi_{\gamma_{\text{sig}}(stp_{alnt})}(\sigma_{\gamma_{\text{sig}}(stp_{alnt})}(T)) \right) \bowtie_{?crs} \left( \pi_{\gamma_{\text{sig}}(stp_{hmnt})}(\sigma_{\gamma_{\text{sig}}(stp_{hmnt})}(T)) \right)
\]

(4.2)

Note that this example can be generalized to joins between multiple star patterns in a natural way.

We implement the R-Type-based query processing model in terms of RAPID+. As shown in Fig. 4.4, SPARQL interface parses a query and ii) *R-Type-based Query Rewriter* transforms SPARQL query expression into the R-Type-based expression.

### 4.3.2 Query Execution

To process R-Type-based query expressions, we need physical operators that implement semantics of logical operators discussed earlier. For this purpose, we introduce a physical operator called \( TG_{-}IndexScan \) that maps to logical operators \( TG_{-}TypeSelection \) and \( TG_{-}Projection \). The algorithm of this operator is shown in Algo. 4. For each Property in a star pattern, the operator probes \( PTG_{\text{Indexes}} \) and collects types that include the Property using \( PTG_{\text{IndexLoader}} \) (line 2-3). A set of types matching a star pattern is computed by intersecting the set of collected types (line 4). Based on the computed types, the operator selectively loads query-relevant triples from type-matching triplegroups (line 5-7).

```
Algorithm 4: TG_{IndexScan}

1. RetrieveTGsForStp (starPattern \( stp \), PTG Index \( ptgi \))
2. foreach Prop \( p_i \) in \( stp \) do
3.       Retrieve types having \( p_i \) from \( ptgi \)
4.       \([stp]\) ← intersect a set of retrieved types ;
5. foreach Typed triplegroup \( tgs \) in \([stp]\) do
6.       retrieve columns representing triples in \( tgs \) using \((off, len)\);
7.       reassemble matching columns, constructing triplegroups matching \( stp_i \);
```
Example 4.3.2. Consider a star pattern $\text{stp}_{\text{at}}$ whose Properties are \{a, t\}. The R-Types that contain Properties a and t can be $\tau_{\text{amnt}}$, $\tau_{\text{ahlmnt}}$, and $\tau_{\text{amnt}}$, $\tau_{\text{ahlmnt}}$, $\tau_{\text{hmnt}}$, respectively. We intersect a set of matching types to determine types matching $\text{stp}_{\text{at}}$, e.g., $\tau_{\text{amnt}}$ and $\tau_{\text{ahlmnt}}$. We then i) retrieve triples in triplegroups having matching types using offset and length augmented in each type and ii) build triplegroups that exactly match $\text{stp}_{\text{at}}$.

In the remaining part of the section, we summarize an overall R-Type-based query processing flow in RAPID+. RAPID+ compiles R-Type-based expression into a set of MR jobs. The first job is responsible for determining matching types and retrieving all relevant triplegroups. The challenge is that RAPID+ needs to estimate appropriate number of mappers based on the size of input before launching MR jobs, but information on matching types is only available once the physical plan is executed using jobs. To solve this issue, RAPID+ exploits pre-computed sizes of matching types from $\text{SZ Indexes}$. Details are omitted due to space constraints, but interested readers can refer to the project website\(^3\).

Once the first job is launched, Hadoop begins to process map-side plan of the job. This map-side plan consists of $\text{TG IndexScan}$s which are responsible for retrieving triplegroups that have matching types. For multi-star-pattern queries, reduce-side plan is subsequently executed in the first job or additional MR jobs are assigned. These jobs execute physical plans consisting of other existing operators such as $\bowtie^\gamma$ for joining triplegroups. For example, the query $Q$ in Fig. 4.1(a) is translated into two MR jobs; the plan in the second MR job includes $\bowtie^\gamma$ for processing joins between triplegroups matching star patterns $\text{stp}_{\text{alnt}}$ and $\text{stp}_{\text{hmnt}}$.

4.4 Semantic Optimization Using Ontological Axioms

4.4.1 Optimizing SemStorm’s Physical Model Using Triple Dematerialization

The introduction section hinted at some semantic optimization techniques which we present in this section. The idea is to leverage ontological axioms in an RDF Schema to identify and dematerialize

\(^3\)http://research.csc.ncsu.edu/coul/RAPID+/SemStorm
‘redundant’ triples (specifically rdf:type triples) in triplegroups, resulting in smaller physical footprint. The benefit of a typed model is that we can reason about redundancies at the level of types rather than individual triplegroups. A related notion of ‘redundant’ subexpressions also can be eliminated from query expressions over R-Types, reducing query processing cost. However,

1. Simply dematerializing redundant triples could introduce ambiguities in determining matches since it may become difficult to distinguish between triplegroups with dematerialized rdf:type triples and similar triplegroups that did not originally contain such triples. Suppose that our example model also had a triplegroup \( tg'' \) that does not contain an rdf:type triple with the Object :Faculty. Superficially, \( tg'' \) would be considered to be of the same type as \( tg_1 \), when in fact they should not be.

2. Not all rdf:type assertions in a triplegroup may be inferable, e.g., a triplegroup \( tg_4 \) contains a non-inferable rdf:type triple whose Object is :crs1. Thus, our typing model should capture the distinction between inferable and non-inferable triples.

Since rdf:type property can be associated all possible classes, the issue of dematerializing them must distinguish the correct rdf:type triples to be dematerialized. Therefore, we refine the rdf:type property by ’promoting’ the associated classes (or Objects) to special Properties, e.g., the pair (rdf:type, :Faculty) is promoted to rdf:type:Faculty (or tFaculty for brevity). Therefore, R-Types that contain such Properties integrate the promoted Properties into their type signatures, e.g., \( \tau_{ahlmntFaculty} \). This refinement of R-Types then allows differentiation of types with dematerialized rdf:type triples vs. similar types that never had such rdf:type triples in the first place. A consequence of this promotion is that the index entries that include the type signatures with promoted rdf:type triples can now be used to discriminate these types in the PTG file.

We can now formalize the semantic optimization that dematerializes ‘inferable’ triples based on the above type refinement.

**Definition 4.4.1.** (D-inferable rdf:type Triples) Let \( H \) be a set of schema triples (triples whose properties are defined in the namespace of RDFS), \( t_{p_1} \) or \( t_c \) be a class-type triple (a triple of the form \( (s, rdf:type, c) \)), \( tg \) be a triplegroup containing triples \( t_1, t_2, t_3, \ldots, t_c \) that have properties \( p_1, p_2, p_3, \ldots, p_i, rdf:type \), respectively. \( t_c \) is then Domain-inferable or D-inferable if

\[
\exists (p_i, rdfs:domain, c) \in H. \text{ Otherwise, } t_c \text{ is non-inferable.}
\]

Based on the definition, we introduce a new type assignment rule that i) dematerializes D-inferable triples in triplegroups and ii) modifies an R-Type by promoting (dematerialized) D-inferable triples in triplegroups to special Properties. Let \( type \) and \( domain \) be abbreviations of rdf:type and rdfs:domain for brevity. Let also \( tg \) be a triplegroup that consists of i) triples with Properties \( p_1, \ldots, p_k \) and ii) D-inferable class-type triples whose Objects are \( c_1, \ldots, c_l \) (its R-type is \( \tau_{p_1 \ldots p_k rdf:type} \)). Given \( tg \) and the entailment rule rdfs2\(^4\), the type of \( tg \) can be reassigned as follows.

\(^4\)https://www.w3.org/TR/rdf11-mt/#rdfs-entailment
Note that strikethrough lines over triples denote that they are dematerialized and promoted as special Properties (denoted by &).

Example 4.4.1. (Dematerializing D-inferable Triples) Fig. 4.5 shows that the class-type triple (:Prs1, rdf:type, :Faculty) in a tg1 is dematerialized and promoted as a special Property, t_Faculty. The type of tg1 is reassigned from τ_ahlmnt to τ_ahlmn&t_Faculty&Person.

The next issue is how to handle R-Types with non-inferable triples. Similar to the D-inferable case, we promote pairs of Properties and Objects in non-inferable triples as special Properties. However, it is possible to cause over-discrimination even among ‘equivalent’ triplegroups with respect to certain queries. In other words, we may end up with too many types that consist of the same Properties with different rdf:types, e.g., τ_ahlmntFaculty&Person, τ_ahlmntFaculty, τ_ahlmntPerson. To relieve the over-discrimination issue, we further partition a set of Properties that consist of types into two subsets: special Properties and other remaining ones. Therefore, triplegroups having such types are also horizontally partitioned into i) non-inferable triples corresponding to special Properties and ii) triplegroups that consist of remaining triples.

Example 4.4.2. (Partitioning Non-inferable Triples) The type for tg4 is modified from τ_hmnrt to τ_hmnrtCourse due to the non-inferable triple (:crs1, rdf:type, :Course) in tg4. τ_hmnrtCourse is then partitioned into two types: τ_hmnr and τ_Course where the latter represents non-inferable triples whose Objects are Course.

4.4.2 Eliminating Redundant Joins from R-Type Expressions

The semantic optimization techniques in the previous section lead to the reassignment of R-Types for triplegroups including class-type triples, therefore we also need to consider how the query processing model can cope with such R-Types. For example, a query expression for the star pattern with D-inferable class-type triple patterns such as stp aloud needs to be reformulated because the semantic optimization
Figure 4.5: Dematerializing a D-inferable triple (:prs1, rdf:type, :Faculty) in a triplegroup $\tau_{tg}$ based on a domain constraint (:1, rdfs:domain, :crs1) and RDFS entailment rule (rdfs2).

techniques dematerialize class-type triples in matching triplegroups such as $\tau_{tg}$ and reassign their R-Types.

We first begin with the scenario in which triplegroups contain D-inferable class-type triples only without non-inferable ones. To dematerialize and rematerialize D-inferable triples in triplegroups, we define two query operators as follows.

**Definition 4.4.2.** (TG_Dematerialize ($\mu^\gamma$) & TG_Rematerialize ($\rho^\gamma$)) Given a triplegroup $tg$ that consists of triples $\{t_1, \ldots, t_i\}$ and D-inferable class-type triples $\{t_{c1}, \ldots, t_{cj}\}$, $\mu^\gamma(tg)$ performs two operations as follows.

i) $tg \rightarrow tg', tg' = tg - \{t_{c1}, \ldots, t_{cj}\}$

ii) modify the type of $tg$, $tg: \tau_{p1 \ldots p_k t} \rightarrow tg': \tau_{p1 \ldots p_k t_{c1 \ldots c_j}}$

$\rho^\gamma$ is the inverse operation of $\mu^\gamma$. Given type signatures of D-inferable class-type triples $\{t_{c1}, \ldots, t_{cj}\}$ and a triplegroup $tg'$, $\rho^\gamma_{\{t_{c1 \ldots c_j}\}}(tg')$ rematerializes class-type triples corresponding to $t_{c1}, \ldots, t_{cj}$ and modifies back the type of $tg'$.

The promotion of type signatures for representing class-type triples brings up additional issues on query processing. While $\sigma^\gamma$ assumes that Objects of all class-type triple patterns are bound, it is possible that queries could have unbound Objects for class-type triple patterns. To handle such queries, we introduce TG_Transform operator that reformulates $\sigma^\gamma$ that represents a star pattern with unbound Objects into a union of $\sigma^\gamma$s with bounded Objects (more precisely, RDFS classes in ontology). We denote unbound Objects of class-type triple patterns as $t^\gamma$.

**Definition 4.4.3.** (TG_Transform ($\theta^\gamma$)) Let $P$ be a set of properties $\{p_1, p_2, \ldots, p_k\}$, $TU$ be a set of properties that consist of rdf:type and unbound Objects $\{t_{z1}, \ldots, t_{z_j}\}$, and $TC$ be a set of $n$ special
Properties that comprise of \texttt{rdf:type} and RDFS classes. Let also $stp$ be a star pattern whose properties are comprised of $P$ and $TU$. Given a $stp$ represented using $\sigma_{P\cup TU}^\gamma$, $\theta_{\gamma,P,TC}^\gamma(\sigma_{P\cup TU}^\gamma)$ returns

$$
\theta_{\gamma,P,TC}^\gamma(\sigma_{P\cup TU}^\gamma) = \sigma_{1}^\gamma \cup \cdots \cup \sigma_{n}^\gamma
$$

where the condition of $\sigma_k^\gamma$ ($k \in [1, n]$) is $P \cup tc_k$, $tc_k \in TC$.

In other words, given a $\sigma^\gamma$ that represents a star pattern having class-type triple patterns with unbound Objects ($t_?$), $\theta^\gamma$ returns a union of the $\sigma^\gamma$s where the unbound Objects in the condition of each $\sigma^\gamma$ are substituted into special Properties that comprise of \texttt{rdf:type} and RDFS classes.

Example 4.4.3. (Reformulating $\sigma^\gamma$ Using $\theta^\gamma$) Assume that we have a variation of star pattern $stp_{hmnt}$ in Fig. 4.1(a), $stp'_{hmnt}$ where the Object of \texttt{rdf:type} is unbound. $\sigma_{hmnt}^\gamma(T)$ that represents $stp_{hmnt}'$ can be reformulated as follows.

$$
\theta_{hmnt,DC}^\gamma(\sigma_{hmnt}^\gamma(T)) = \sigma_{hmnt t_{\text{Course}}}^\gamma(T) \cup \sigma_{hmnt t_{\text{Faculty}}}^\gamma(T)
\cup \cdots \cup \sigma_{hmnt t_{\text{Course}} t_{\text{Person}}}^\gamma(T).
$$

where $DC$ is the powerset of type signatures representing all D-inferable class-type triples, \{\text{\texttt{t}Course}, \text{\texttt{t}Person}, \text{\texttt{t}Faculty}, \text{\texttt{t}Course} \text{\texttt{t}Person}, \ldots, \text{\texttt{t}Course} \text{\texttt{t}Person} \text{\texttt{t}Faculty}\}.

Note that the operators introduced in this section allows for producing answers equivalent with the ones over R-Type model without the semantic optimizations, which can be proved as follows.

Lemma 4.4.1. Query processing over R-Types with dematerialized D-inferable triples is lossless.

Proof. Let $stp$ be a star pattern that includes class-type triple patterns with bound Objects ($t_{p_{c_1}}, \ldots, t_{p_{c_l}}$) and unbound Objects ($t_{p_1}, \ldots, t_{p_l}$), and $c_{e_1}, \ldots, c_{e_l}$ and $t_{1}, \ldots, t_{l_1}$ be class-type triples respectively matching $t_{p_{c_1}}, \ldots, t_{p_{c_l}}$ and $t_{p_1}, \ldots, t_{p_l}$. We prove the losslessness in terms of boundedness of D-inferable triple patterns in $stp$.

Case 1: Bound D-inferable class-type triple patterns. In this case, only unbound fields of class-type triple patterns are Subjects and dematerialized triples have equivalent Subjects as the Subject of other triples in triplegroups. Hence, Subjects of dematerialized triples are derivable without rematerialization operations and the semantics of $stp$ is the same as before in (5.1).

Case 2: Unbound D-inferable class-type triple patterns. There are two classes of unbound fields in class-type triple patterns: Subjects and Objects. While Subjects can be derivable as the Case 1, the Objects are not. Thus, this case requires explicit rematerialization of matching triples. We therefore transform $\sigma^\gamma$ using $\theta^\gamma$ to ensure to capture all types matching $stp$ with unbound Objects in $t_{p_{1}}, \ldots, t_{p_{l_1}}$. We then apply $\rho^\gamma$ for rematerializing matching D-inferable triples to capture all bindings for unbound
Objects of class-type triple patterns in \( stp \). Let \( DC \) be the powerset of type signatures representing all D-inferable triples. The query expression for this case can be represented as follows.

\[
[[stp]] = \pi_t^\gamma_1,\ldots,\pi_t^\gamma_l (\sigma^\gamma_{\text{sig}(stp)}(T)))
\]

The remaining issue is how to process non-inferable class-type triples. Because such triples are horizontally partitioned from triplegroups, we similarly partition a star pattern into i) class-type triple patterns and ii) remaining ones, and then interpret them as separate star patterns. Note that we also need to join back i) matching class-type triples and ii) remaining ones to ensure the correctness. Similar to the case of D-inferable triples, this processing does not cause any information loss or introduce any spurious results, which can be proved as follows.

**Lemma 4.4.2.** Query processing over R-Types with partitioned non-inferable triples is lossless.

**Proof.** Let us assume that we reuse the notation of variables defined in the the proof of Lemma 4.4.1. Similar to the proof of Lemma 4.4.1, we prove losslessness in terms of boundedness of Objects of class-type triple patterns in a star pattern \( stp \).

**Case 1. Bound non-inferable class-type triple pattern.** Let \( stp' \) be a star pattern where all class-type triple patterns are removed (i.e., \( stp - \{tp_{\gamma_1}, \ldots, tp_{\gamma_l}\} \)). In this case, joins between partitioned class-type triples matching \( tp_{\gamma_1}, \ldots, tp_{\gamma_l} \) and triplegroups matching \( stp' \) ensures to compute all triplegroups matching \( stp \). The query expression for this case can be given as follows.

\[
[[stp]] = \pi_t^\gamma_{\text{sig}(stp)}(\sigma^\gamma_{\text{sig}(stp)}(T)) \vee \gamma
\]

**Case 2. Unbound non-inferable class-type triple pattern.** Let \( SC \) be a set of all types representing (partitioned) non-inferable triples. This case only differs with the Case 1 in that the expression includes \( \theta^\gamma \) to ensure that the expression captures all types matching \( tp_{\gamma_1}, \ldots, tp_{\gamma_l} \) from \( SC \). The expression is therefore

\[
[[stp]] = \pi_t^\gamma_{\text{sig}(stp)}(\sigma^\gamma_{\text{sig}(stp)}(T)) \vee \gamma
\]

**Example 4.4.4.** (Query Processing over R-Type Model) Fig. 4.6 shows how star patterns \( stp_{\text{alnt}} \) and \( stp_{\text{hmnt}} \) in Fig. 4.1(a) are processed over the R-Type model with semantic optimization techniques. \( stp_{\text{alnt}} \) and \( stp_{\text{hmnt}} \) include D-inferable and non-inferable triple patterns (\( t_{\text{Faculty}} \) and \( t_{\text{Course}} \)), respectively. We assume that we already construct triplegroups, assign types to them, and dematerialize D-inferable triples in the triplegroups using \( \mu^\gamma \) as a pre-processing. In the stage of query planning, we
rewrite star patterns based on the inferability of class-type triples and launch the first MR job that selects matching types using $\sigma^\gamma$, e.g., (i) $\tau_{almn\&tFaculty}\&tPerson$ and $\tau_{almn\&tFaculty}$ for $stp_{almn}$ and (ii) $\tau_{hmnr}$ and $\tau_{tCourse}$ for $stp_{hmnr}$. In the query processing stage, $\tau_{hmnr}$ is joined with $\tau_{tCourse}$ using a map-side join and $\pi^\gamma$ is applied over matching triplegroups to produce triplegroup that exactly match star patterns. Finally, triplegroups matching each star pattern are joined using $\Join^\gamma$.

We finally note that the R-Type model along with the proposed optimizations is implemented on recent Hadoop frameworks. The left side of Fig. 4.7 shows conceptual execution flows and the right side illustrates more detailed workflows with the components of the recent Hadoop framework such as Apache Tez (a distributed execution framework) [4] and Apache Yarn (a resource management layer for Hadoop clusters) [3]. Logical plans for pre-processing and query processing are translated and compiled into Tez jobs which represent data flows as DAGs. These DAGs are then executed under the supervision of Yarn which assigns and manages resources in clusters.

### 4.5 Empirical Evaluation

#### 4.5.1 Testbed Setup

The goal of our evaluation was to compare the performance of extended RAPID+ (i.e., RAPID+ with SemStorm) with other storage schemes that were discussed earlier.
Benchmark Dataset and Queries. Our testbed consists of two RDF benchmark datasets – Lehigh University Benchmark (LUBM), synthesized based on an ontology schema from academic domain, and DBPedia SPARQL Benchmark (DBPSB), generated based on DBPedia datasets. The two datasets show contrasting characteristics as summarized in Table 5.1. While LUBM has small number of large-sized Properties / types, DBPSB has large number of sparse Property / types.

Testbed queries were adapted from LUBM and DBPSB benchmarks to incorporate scenarios with \texttt{rdf:type} optimizations. Query selectivities were varied by binding Object fields of non-\texttt{rdf:type} triple patterns (high-selectivity, denoted with postfix \texttt{a}), and evaluating same query with unbound Object (low selectivity marked with \texttt{b}). Queries with significantly high execution time are marked using \texttt{≈} after confirming that correct answers were produced. Approaches that failed to produce answers due to errors such as MR job failures, are marked using a red-colored \texttt{×}. All evaluated queries are available on the project site.

Evaluated Approaches. We evaluated two implementations of RAPID+ to measure the effectiveness of column-oriented layout used in PTG file format: i) \textit{SemStorm(R)} that stores triplegroups as a set of rows consisting of Subjects and Objects and ii) \textit{SemStorm(C)} that stores triplegroups as a set of columns as described in Section 4.2.2.

We simulated approaches using Hive 0.12.0 with a TEXTFILE format to avoid effects of various storage-side optimizations (e.g., compression schemes in ORC). Triples were loaded into HDFS and imported into a ternary table $T$, which was then partitioned using the following three approaches:

1. \texttt{Hive(Sub-Bkt)} simulated hash-partitioning of triples on Subject. Table $T$ was bucketed on Subject
(full MR job) and triples in each bucket were sorted to enable map-side sort-merge joins. Each bucket
corresponds to a single file in HDFS.

2. **Hive(VP)** simulated the vertical partitioning approach, which partitions table $T$ based on Property
types using a map-only job per Property type. This approach produces $nk$ physical files when $n$
Property relations exist on a $k$-node cluster.

3. **Hive(VP-Bkt)** is a hybrid approach of Hive(Sub-Bkt) and Hive(VP). This approach buckets Property
relations on Subject to simulate the approach of CliqueSquare.

In general, **Hive(VP-Bkt)** is preferred over **Hive(Sub-Bkt)**, since the former selects only relevant buck-
eted Property relations, resulting in less disk and network I/O. However, **Hive(VP-Bkt)** is not practical
for datasets such as DBPSB with high number of Property types, making it challenging to determine
appropriate number of buckets, apart from the overhead of user having to write large number of table
generation statements (each executed in 1 MR job). Hence, we only evaluated **Hive(Sub-Bkt)** in the case
of DBPSB.

We also evaluated other recent Hadoop-based RDF query processing systems as follows.

4. **H$_2$RDF+** implements a multi-indexing scheme introduced in RDF-3x on HBase$^5$. Each HBase table
corresponds to a distributed sorted key-value map, thus 6 tables are assigned to store six permutations
of RDF triples as keys with null values.

5. **CliqueSquare** employs a co-partitioning scheme that co-locates all triples having the same Subject,
Property, or Object within the same node, enabling construction of clique subgraphs using map-side
joins.

Semantic Hash Partitioning [54] and EAGRE [86] were not publicly available for evaluation. HadoopRDF [44]
was not evaluated since $H_2$RDF$^+$ [63] and **CliqueSquare** [34] are shown to outperform it. Though we
evaluated only Hadoop-based processing systems, we included RDF-3x for completeness, which is the
state-of-the-art centralized RDF query processing system.

**Cluster Configuration.** Evaluation was conducted on 20-node and 80-node Hadoop clusters in
VCL$^6$, with each node equipped with Xeon dual core x86 CPU (2.33 GHz), 4GB RAM, and 40GB
HDD. While cluster sizes may appear larger than necessary, the 80-node cluster made available 1.6TB
disk space (20GB per node) which was required to store the intermediate data materialization, e.g.,
pre-processing of the 450GB LUBM dataset materialized 2-3 times of the input size. For RDF-3x, a
dedicated single physical node was used, with Xeon quad-core CPU (E5410, 2.33GHz), 40GB RAM,
and 4TB HDD. All results were averaged over three or more trials. To ensure fairness in comparison, we
report execution time of MR jobs to to minimize effects of irrelevant performance factors and prepara-
tion steps. For example, we do not consider the post-processing step in $H_2$RDF$^+$ required to decompress

---

$^5$https://hbase.apache.org
$^6$http://vcl.drupal.ncsu.edu
### 4.5.2 Evaluation Results

#### Pre-processing Input Datasets

Both $H_2RDF+$ and CliqueSquare failed to import DBPSB datasets due to data format issues caused by malformed triples. Additionally, CliqueSquare generated too many files, i.e., approx. 670k files were generated on the 20-node cluster burdening the namenode with meta information maintenance. We expect that the number of physical files in HDFS would increase further with the increase in number of nodes, likely to cause memory issues.

**Storage Requirement.** The aggressive compression/encoding schemes in $H_2RDF+$ and RDF-3x reduce disk space usage. For SemStorm and all Hive approaches, we shortened URIs with pre-defined prefixes, producing QNames for URIs, e.g., rdf:type instead of `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`. In general, SemStorm required less amount of storage compared to all Hive approaches by avoiding duplication of common Subjects in triplegroups. Reduced storage requirement in SemStorm can also be attributed to optimization of rdf:type triples, which reduced the number of materialized triples by 17.4% and 6.2% for LUBM and DBPSB, respectively. Hive(VP-Bkt) required less storage when compared to Hive(Sub-Bkt) since the former materializes only Subject-Object components (no Property types). CliqueSquare showed a high storage requirement due to replication of uncompressed triples for building three types of partitions across nodes.

**Pre-processing Time.** Hive(VP) and Hive(VP-Bkt) required multiple MR jobs to retrieve unique Properties / Classes in datasets and generate bucketed tables per Property. $H_2RDF+$ took a significant amount of time for dictionary-encoding datasets as well as building six indexes and aggregated index statistics. Similar trends were also observed in RDF-3x. SemStorm was mostly faster than other approaches due to a simple pre-processing step that uses a single MR job. CliqueSquare also used a single MR job, but was slow due to its high storage requirement. The details of a storage requirement and pre-processing time are available at our project site.

#### Varying Selectivity of Graph Patterns

This task studies the impact of selectivities of different Property types, e.g., Property :name is popular across classes while Property :subOrganizationOf is only associated with class Organization. Query $L1$ consists of a single triple pattern with Property :name, while queries $L2-L4$ include a sec-
ond triple pattern with additional Properties that show increasing selectivity (decreasing popularity). Execution times for different approaches are shown in Fig. 4.8. Overall, Hive(VP-Bkt) was faster than Hive(VP) due to less expensive map-side sort-merge joins. SemStorm(R) was in general slower than SemStorm(C) due to the limitation of a row-oriented file format, i.e., scans of irrelevant values in triple-groups. Hive(VP), Hive(VP-Bkt), and H2RDF+ performed best for query L1 that scans a single Property relation :name, while SemStorm scanned all types including the Property :name. Note that H2RDF+ produced a sub-optimal plan with a centralized approach for query L1b. Queries L2-L4 demonstrate the benefit of SemStorm due to selective scans of matching types leading to a reduction in disk read I/O. Other triple-based approaches including RDF-3x, H2RDF+, and CliqueSquare tend to increase its execution time for L1-L3. All approaches were fast to process query L4, which involved a highly selective Property. Queries L2-L3 were extended (L5-L6) to study the impact of denser star patterns. The gap between Hive(VP) and SemStorm further increases due to scans for irrelevant triples in Hive(VP).

Two-star pattern queries included query L7 (stars with same set of Properties) and query L8 (stars differed in some Properties). In the case of L7, SemStorm required the same set of matching types for both stars, enabling shared scans. Even for L8, SemStorm shared scans of types that match both star patterns. SemStorm(R) was slower than Hive(VP-Bkt) due to scans of irrelevant values in triple-groups. SemStorm(C) overcomes this issue by scanning only required values in matching (typed) triple-groups.

While RDF-3x showed impressive performance for high selective queries such as L7a, its performance degraded for queries such as L1a and L1b which return approximately 400k and 223m tuples, respectively. As observed in [44, 85], RDF-3X has an overhead of scanning a wide range of multi-indexes while other approaches benefit from parallel scans of horizontally partitioned datasets. H2RDF+ was able to process high-selectivity queries such as L7a very quickly using its centralized approach (Marked with ’C’). However, H2RDF+ showed poor performance for low-selectivity queries, mainly due to the overhead of scanning a significant amount of irrelevant triples. Additionally, multi-star queries were processed using a left-deep plan that translated to a sequence of jobs, one for each join operation [34]. For this reason, many jobs were often sequentially processed, resulting in a significant amount of execution time. In the case of CliqueSquare, we observed many failures due to the OOM issue. We speculate that such errors could be avoided with the use of compression schemes such as QName.

Varying Density of Star Patterns

This task considers queries D1 - D6, with increasing number of triple patterns in a star pattern, starting with base query D1 that contains a single triple pattern. Fig. 4.9(a) shows that execution times of Hive(Sub-bkt) increases with increasing density of star patterns due to high HDFS reads (refer to Fig. 4.9(b)). In the case of SemStorm, there is a decrease in the execution time and the amount of disk reads for D1-D3 due to a decrease in the number of matching types. For remaining queries, the number of types matched by SemStorm were comparable, and hence a similar performance trend is observed.
Performance of RDF-3x increased for low query selectivities due to its exhaustive indexing. Though it was not practical to process DBPSB using Hive(VP/VP-Bkt) due to the large number of Properties / Classes (bucketing ≈ 200k of them would take several days), for demonstration purposes, a subset of relations were created for 11 Properties and 8 Classes used in queries D1-D6. Fig. 4.9(c) shows the accumulated processing times of tested queries including time for pre-processing and execution of all queries. Note that Hive(VP) and Hive(VP-Bkt) took more than 2.5K seconds to pre-process the subset, while SemStorm processed ALL of them in the same time. Additionally, SemStorm took the least amount of time to execute all queries.

**Answering Negative Queries**

While exploring large datasets with a substantial number of Properties and Classes, it is common to encounter queries that may not produce any results. A set of queries were designed to test such negative scenarios – queries L14 (single star, LUBM) and D23-D24 (single star, DBSPB). Fig. 4.10(a) and Fig. 4.10(b) shows that all other approaches returned 0 results after spending time for scanning Property relations / exhaustive indexes and discarding unmatched results using expensive joins. On the contrary, SemStorm was able to check whether matching type existed by accessing PTG indexes only (without
accessing PTG files that contain triplegroups) and therefore halted the execution of MR jobs quickly. This allows SemStorm to be up to 500 times faster than other approaches.

![Graph comparing execution times of different approaches](image)

**Figure 4.10:** Answering negative queries (a) LUBM (b) DBPSB

**Varying Number of D-inferable rdf:type Patterns**

This task studied the effectiveness of semantic optimizations. We omit Hive(Sub-Bkt) / SemStorm(R) since they were surpassed by Hive(VP/VBkt) / SemStorm(C), for all previous queries.

**Single D-inferable rdf:type Triple Pattern.** Queries L9 and L10 contain a single rdf:type triple pattern, and a non-type triple pattern that can be used to infer the rdf:type triple pattern. For reference, we also include queries that contain non-inferable rdf:type triple patterns, i.e., L12 and L13. Other approaches scanned both rdf:type and non-rdf:type Property relations, e.g., L9 reads :teachingAssistantof (694MB) + tteachingAssistant (1,295MB) = 1,989MB for Hive(VP). However, SemStorm only reads triplegroups from matching types, which is particularly effective for low-selectivity queries. Query L11 that joins two star patterns (L9 $\bowtie$ L10) also followed similar trends.

**Multiple D-inferable rdf:type Triple Patterns.** Since LUBM does not contain more than one explicit rdf:type triple per triplegroup, DBPSB was used to study cases with multiple D-inferable and non-inferable triple patterns. Base queries D9 and D14 were extended to include two D-inferable triple patterns (D10 and D15) and two non-inferable ones (D11 and D16), respectively. We also evaluated additional queries such as D12 and D17 (with one D-inferable and one non-inferable one) and D13 and D18 (with two D-inferable and two non-inferable ones), along with two star pattern queries (D19-D22) with D-inferable and non-inferable triple patterns whose results are shown in Fig. 4.12(c). The overall trend in Fig. 4.12(a-b) shows that the performance of Hive-based approaches degraded as we added rdf:type triple patterns, while SemStorm incurred less HDFS reads and performed better for queries with D-inferable triple patterns.

49
Figure 4.11: Evaluating queries with rdf:type triple patterns using LUBM datasets (Univ-20k, 80-node cluster)

Figure 4.12: A performance comparison of queries with multiple (non-)inferable rdf:type triple patterns (DBSPB-200, 20-node cluster)

4.6 Chapter Summary

In this chapter, we present an R-Type model based on a natural equivalence relation on R-Typed triple-groups. To exploit this typing for improving efficiency of graph pattern matching, we discuss (i) an autonomic scheme that induces a typing model in a user-independent manner, (ii) a type-aware query processing model for translating queries into expressions on the typed model, and (iii) an indexed file organization that uses a polymorphic, nested-column file format to support an efficient discriminatory access based on types. We also introduce semantic optimizations that exploit ontological axioms to de-materialize redundant triples in storage models as well as eliminate redundancies in query expressions.
Chapter 5

Optimizing Processing of Complex Queries on RDF Data Using SemStorm

5.1 Motivation

One of the characteristics of Big Data is the heterogeneity or so-called ‘variety’ which analysis techniques need to account for to provide a holistic and integrated perspective of data. A holistic and integrated perspective of data is a key demand of real-world data analysis scenarios in many domains due to the necessity of considering different types and sources of data. A trend that has emerged as a result of this demand is the use of ‘formal ontologies’ for semantic data integration. One example of semantic integration based on ontologies is Uniprot [20] which is a central hub for functional information on proteins. It comprises of 16 datasets (approx more than 1TB in N-Triples) that includes core datasets like amino acid sequence, protein names or descriptions, taxonomic data and citation information as well as a lot of annotation information from various widely accepted biological ontologies, classifications and cross-references.

However, realizing the benefit semantic or ontological data integration requires support for semantic or ontological querying that must account for the need to perform such logical inferences as part of query answering. Unfortunately, effective techniques for efficient analytics of data represented in these forms are still needed, leaving the potential benefits of semantic data integration unrealized. This requires the consideration of implicitly represented data that must be reasoned about using ontological axioms. Most existing reasoning techniques employ semantic reasoners that derive implicitly represented statements from explicit statements and ontological axioms using forward-chaining [79, 57, 51], backward-chaining [80, 33], or hybrid ones [57, 33]. However, these approaches tend to be inherently sequential, making them difficult to parallelize and scale up to Big Data.

As alternative approaches to existing techniques, one of the recent directions is to (i) express ontologies using lightweight ontology languages that is first-order rewritable and (ii) reformulate ontological
(conjunctive) queries with respect to the relevant ontological axioms, generating union of conjunctive queries (UCQs) [27]. For example, suppose that we want to know all E-Coli K12 UniProt entries (including strains) and their amino acid sequences (Query 3 in Uniprot SPARQL endpoint\(^1\)). The Uniprot core dataset can be used to find matches for “amino acid sequences for E-Coli K12 proteins”. However, by expanding based on the relationships in the taxonomy dataset, all sub-taxons of the E-Coli K12, e.g., E-Coli (strain K12 / DH10B) can also be considered. Therefore, during query processing, the pattern will be expanded to include alternatives e.g., UNION “amino acid sequences for E-Coli (strain K12 / DH10B)” UNION ..., etc.

The excitement with this approach is that it essentially reduces the ontological querying problem to a database problem, i.e., the evaluation of UCQs can be delegated to existing database systems that support the processing of conjunctive queries represented using SQL or SPARQL. While the FOL-rewritability property is not universal to all ontology languages, it holds for a good subset of popular ontology languages such as RDFS, OWL2 QL and (most of) OWL2 EL. However, practical implementations of this approach for Big Data scenarios are yet to emerge. The major challenge is that the query rewritings generated from ontological queries are far more complex than existing systems can cope with.

**Motivating Example.** Consider the ontological query \(gp\) which retrieves the list of students who takes some courses. Fig. 5.1 shows that \(gp\) is rewritten into a query with additional star patterns inferred from the original ones, i.e.

\[
gp' = (stp_1 \bowtie stp_2) \cup (stp'_1 \bowtie stp_2) \cup (stp''_1 \bowtie stp_2) \cup \ldots
\]

\(^1\)http://sparql.uniprot.org
where \( stp_1 \) and \( stp_2 \) denote possible reformulations of the first star pattern \( stp_1 \). \( stp_2 \) also can similarly be expanded. Let us assume 10 axioms for class :Student in \( stp_1 \) and :Course in \( stp_2 \) respectively. Then each star pattern would be reformulated into 10 different rewritings respectively, which leads to 100 reformulated graph patterns in \( gp' \). Each reformulated pattern consists of 4 triple patterns that involve 4 selections and 3 joins, therefore processing \( gp' \) would require 400 selections and 300 joins. Note that triple pattern \( tp_2 \) and \( tp_4 \) are included in every union branch and repeatedly processed. Processing queries with many operations in a sequential manner would require a significant execution time, therefore distributed processing platforms such as Hadoop can be considered to parallelize the process of operations. However, the challenge is that each join involves re-partitions of intermediate data across nodes in a cluster using a significant amount of disk and network I/Os, while processing \( gp' \) requires 300 joins. As an alternative approach, let us consider employing special operators for optimizing the processing of ontological queries. If we could assume special operators that take star patterns as parameters and process relevant star subgraphs, we could rewrite the \( gp \) into a disjunctive expression as follows:

\[
gp' = \sigma_{\gamma}^{\gamma + \sigma} \left( stp_1 \vee stp_2 \vee \ldots \right) \wedge \gamma + \sigma_{\gamma}^{\gamma + \sigma} \left( stp_2 \vee stp_2 \vee \ldots \right)
\]

where (i) \( \sigma_{\gamma}^{\gamma +} \) selects all star subgraphs matching original star patterns and their reformulations and (ii) \( \wedge \gamma + \) joins relevant subgraphs. Note that unions in UCQs are pushed down and implicitly captured as disjunctions, which allows reducing the number of joins and repartitions steps into 1.

As shown in this example, rewriting into disjunctive expressions can be beneficial for optimizing the execution of ontological queries. These special operators could be built on aggregate data model and algebras shown in the previous chapters, but a number of challenges arises in practice: (i) reformulating star patterns often lead to irregular patterns which are more complex than star patterns and therefore some reformulations cannot be linked using disjunctions and (ii) some reformulated patterns are highly unselective, which requires evaluating a large number of triples in data.

**Contribution.** To solve aforementioned issues, we investigate reformulation and execution strategies of ontological queries. We begin by optimizing unselective patterns in UCQs and further exploring the possibility of adapting existing rewriting techniques such as union-pushdown [30] and multi-query optimizations [53] for optimizing the execution of UCQs. We also discuss the limitation of these techniques such as requiring additional operators and disk/memory spaces for processing a larger amount of intermediate data produced. To overcome these limitations, we present alternative strategies based on aggregate data and query processing models which allows (i) reducing the complexity of UCQ expressions thereby minimizing the number of operators and (ii) achieving a co-evaluation of reformulated star patterns and an early pruning of irrelevant tuples. Specifically, the contribution of this paper is threefold:

- A comprehensive comparison study of query reformulation strategies for optimizing the execution of
UCQs: (i) rewriting UCQs using unbounded triple patterns and disjunctive predicates, (ii) pushing unions under joins down using union-pushdown, and (iii) adapting multi-query optimization technique (MQO) that identifies common subexpressions across union branches of UCQs and shares the execution of the subexpressions.

- **Semantic reformulation techniques that rewrite chain-joins in UCQs using ontological axioms, reducing the complexity of graph patterns.** These reformulation techniques are closely integrated with a type-based query processing model which allows co-evaluating reformulated graph patterns as well as minimizing the number of operators and re-partition processes in distributed contexts.

- An extensive evaluation conducted on synthetic datasets with customized query sets. The evaluation results show that our approach was up to 25X faster than existing approaches, using minimal footprint of disk/network I/Os.

### 5.1.1 Related Work

Optimizing UCQs was mainly studied in the theoretical context of optimizing its rewriting such as reducing the size of rewriting by eliminating duplications [37] or employ other expressive languages such as Datalog [67, 32]. Recently some approaches [25] have adapted relational cost-based optimization techniques for reordering the execution of subexpressions to yield lower cost plans. However, existing cost-based model are targeted centralized execution whereas distributed execution would prioritize different criteria in order to avoid re-partition steps which incurs network data transfer costs. Optimization of UCQs execution is closely related to disjunctive query processing, i.e., disjunctive queries can be translated into a normal form, e.g., DNF (Disjunctive Normal Form) where disjunctions are processed using union operations [30], or CNF (Conjunctive Normal Form) where disjunctive predicates remain in parameters of operators [46, 77, 76]. In this component, we study such rewritings of UCQs and evaluate them using distributed query processing platforms such as Hive on Hadoop.

### 5.2 Optimization Strategies for Processing UCQs in Distribute Contexts

As alluded earlier, evaluating UCQs often involves a significant processing overhead which is caused by processing many unions and joins. One possible strategy is reducing the overhead from processing unbound triple patterns generated by the rule R4 and R5, e.g., the red-colored one in Fig. 5.2(a).

#### 5.2.1 Increasing the Selectivity of Graph Patterns in UCQs

If we take a closer look on graph patterns in UCQs, the unbound triple patterns are always joined with other ones. For example, $\langle ?s \ ?p \ ?x \rangle$ from R4 is joined with $\langle ?p \subseteq_{sp} ?r \rangle$ via a variable $?p$. Therefore, if we could first evaluate all other triple patterns in a query and compute the bindings of these join variables in
advance, we could rewrite the unbound patterns using the bindings, i.e., substituting join variables in the unbound triple patterns into bindings. We can then evaluate these reformulated patterns, which are more selective. We note that multiple bindings can exist for the join variable and therefore a union is added to capture all bindings, i.e., this rewriting will produce a union of the bound triple patterns. We also note that all other bound triple patterns in reformulated graph patterns are schema triple patterns whose Properties are defined in the namespace of RDFS, e.g., \( \langle ?c \subseteq \text{sc} ?e \rangle \), \( \langle ?p \subseteq \text{sp} ?r \rangle \), \( \langle ?r \subseteq \text{r} ?c \rangle \). Therefore, we can selectively evaluate such schema triple patterns over the schema closure (rather than evaluating them over whole RDF graphs), resulting the elimination of schema triple patterns in queries. A union of bound triple patterns also does not need to be evaluated over schema closure but data (non-schema) triples. Based on this idea, we formalize the rewriting technique as follows.

**Definition 5.2.1. (REWRITE)** Let \( s^+ \) be a schema closure generated by schema \( s \). Let also \( \mu(\?x) \) be an assignment generated from the evaluation of \( \{tp\}^+ \) over \( s^+ \), \( \{(\?x \leftarrow x_1), ..., (\?x \leftarrow x_n)\} \). Given a triple pattern \( dtp \in \{tp\}^+ \) and \( \mu(\?x) \), **REWITE** \( dtp, \mu(\?x) \) returns a set of triple patterns linked by unions,

\[
\{dtp^n_1 \cup \ldots \cup dtp^n_n\}
\]

where \( dtp^n_i \) denotes the result of substituting variable \( \?x \) in \( dtp \) into the value in assignment \( x_i \). Essentially, assuming that we have a graph pattern \( stp \), we first evaluate schema triple patterns over a schema closure and compute the mappings for \( \?x \), which are \( x_1 \) to \( x_n \). The **REWITE** then produces unions of the triple patterns which are identical with the triple pattern \( dtp \in \{tp\}^+ \) except that the variable \( \?x \) in \( dtp \) is substituted into the mappings \( x_1 \) to \( x_n \).

**Example 5.2.1. (Rewriting Graph Pattern Generated by R4)** Let \( \{tp\}^+ \) be a set of triple patterns generated by reformulating \( \langle \?s \text{ rdf:type c} \rangle \) using the rule R4,

\[
\{tp\}^+ = \{\langle c \subseteq \text{sc} ?e \rangle, \langle ?p \subseteq \text{sp} ?r \rangle, \langle ?r \subseteq \text{r} ?c \rangle, \langle s ?p \?x \rangle\}.
\]

Fig. 5.2(a) shows the execution plan for the \( \{tp\}^+ \), which includes the subplan that scans and joins all triples (colored as red). Fig. 5.2(b) illustrates the process that rewrites the initial plan using the **REWITE**. In \( \{tp\}^+ \), \( \{tp\}^+_s \) is \( \{\langle c \subseteq \text{sc} ?e \rangle, ..., \langle ?r \subseteq \text{r} ?c \rangle\} \) and \( \{tp\}^+_d \) is \( \{\langle s ?p \?x \rangle\} \). The common variable exists between \( \{tp\}^+_s \) and \( \{tp\}^+_d \) is the \( ?p \). Let us assume that evaluating \( \{tp\}^+_s \)
Figure 5.2: (a) An execution plan for a reformulation of the triple pattern $tp$, $\langle \text{?s rdf:type ub:Student} \rangle$ and (b) the rewriting process that generates more selective plans.

over $s^+$ returns the mappings of $?p$, $\mu(?p)$, such that $\{(?p \leftarrow p_1, ..., (?p \leftarrow p_n)\}$. $dtp_d$, a data triple pattern in $\{tp\}^d$, is then $\{(?s \ ?p \ ?x)\}$. Finally, the function $\text{REWRITE} (dtp_d, \mu(?p))$ rewrites $dtp_d = \{(?s \ ?p \ ?x)\}$ into a union of triple patterns which are identical to $dtp_d$ but $?p$ in those triple patterns are substituted into the values $(p_1, ..., p_n)$, i.e.,

{\langle ?s p_1 ?x \rangle \cup ... \cup \langle ?s p_n ?x \rangle}

The right side of Fig. 5.2(b) shows the resulting query plan for the union of the triple patterns.

Note that the rules shown in the Section 2 (R3 to R5) generates three graph patterns connected by a single union and therefore applying $\text{REWRITE}$ for each graph pattern generates three independent subplans that consist of a union of triple patterns. The unions in these subplans can be merged as a single union as optimizations.

5.2.2 Rewriting Strategies for Optimizing the execution of UCQs

The previous rewriting technique allows increasing the selectivities of graph patterns in UCQs but the number of graph patterns increases as a trade-off. This again could cause an overhead because the number of repeated evaluations for common subexpressions such as non-rewritable triple patterns (e.g.,
Rewriting as Graph Patterns with Disjunctive Predicates

The first rewriting strategy is to reformulate UCQs as a disjunctive query, i.e., graph pattern with disjunctive predicates. Specifically, a triple pattern \( \langle ?s \text{ rdf:} \text{type} c \rangle \) which is expanded into triple patterns \( \{ \langle ?s \text{ p1} ?x \rangle, \langle ?s \text{ p2} ?x \rangle, \ldots \} \) can be rewritten into a triple pattern with additional variables and filter conditions as follows.

\[
\langle ?s \ ?p \ ?o \rangle \ \text{FILTER} \ ((\text{regex}(?p, 'p1') \ |\ | \text{regex}(str(?x), 'p2') \ |\ | \ldots)
\]

Fig. 5.3(a) shows the reformulated plan based on this approach. We note that other query constructs such as VALUES also can be used to build similar execution plans. The main advantage of this approach is to enable the shared evaluation, i.e. for each triple pattern, a single scan of triples can evaluate all relevant filter conditions at once. This approach therefore could reduce the frequencies of re-partitioning intermediate data, which would be advantages in distributed context.

Limitation. One potential drawback is that efficient access paths such as indexes or vertical partitions
cannot be utilized to selectively access matching triples. Therefore, processing a single triple pattern always involves a full scan of triples. For this reason, many query optimizers transform disjunctive queries into UNIONS to avoid full scans of tuples, i.e., OR Expansion [13].

**Rewriting Using Union-Pushdown**

The second strategy is to adapt union-pushdown techniques to push unions under joins to avoid an repeated evaluation of common subexpressions. Fig. 5.3(b) shows the query plan based on this approach. Triples matching $tp_1$ and all of its reformulations such as $tp'_1$ and $tp''_1$ are first merged using unions and then joined with other triple patterns. $tp_3$ and its reformulation is similarly processed. The potential benefits of this strategy could be several: In addition to avoiding repeated evaluations of common expressions, this technique significantly simplifies the plan, reducing the overall number of operators and re-partitions of intermediate data.

**Limitation.** As this technique first merges all triples matching triple patterns using unions before pruning out irrelevant triples using joins, a large intermediate data can be produced. The intermediate data later then need to be partitioned, shuffled, and sorted across nodes for joins, which could become a significant performance bottleneck in a distributed context.

**Rewriting Using Multi-Query Optimizations**

The final approach to be considered is to adapt multi-query optimization techniques (MQO), i.e. identify common (overlapped) subexpressions in multiple queries, execute them once, and share the intermediate results for processing other non-common subexpressions. UCQs can be similarly processed based on the MQO technique [53] developed in the context of SPARQL. Fig. 5.3(c) shows the result of reformulating gp using the MQO. Common subexpressions across union branches are placed outside optional clauses (e.g., $tp_2$ and $tp_4$) and remaining uncommon expressions are added under optional clauses (e.g., each reformulations of $tp_1$ and $tp_3$). The first two possible combinations of union branches can be translated into two optional clauses $opt_1$ and $opt_2$, e.g., $opt_1$ denotes a set of triple patterns $tp'_1$ and $tg_3$ grouped under a optional clause. These optional clauses are then processed using left outer joins.

**Limitation.** It is possible that some results of the reformulated queries would match multiple clauses or would not match any optional clauses. To cover such cases, we would need the post-process that eliminates spurious results and distributes results across multiple branches. In addition, this approach could produce large intermediate results because it essentially replaces unions into outer joins which do not fully eliminate irrelevant results. Applying multiple outer joins also generates wide relations while managing such relations is often challenging in practice [11, 1].
Discussion

We so far overview possible rewriting strategies of UCQs. While these approaches enable sharing of the evaluation of common subexpressions in some degrees, a large amount of intermediate data is often produced as a result, which leads to a significant amount of network and disk I/Os. A large portion of intermediate data often consists of irrelevant tuples which are later pruned using joins. Therefore, if we could prune irrelevant data as early as possible while sharing the evaluation of common subexpressions, we could minimize I/Os for processing UCQs which would enhance overall processing performance.

One promising strategy to achieve the early pruning is to adopt algebraic and type-based semantic optimization techniques introduced in Chapter 4. The R-Type system and its query processing model allow an early pruning of irrelevant intermediate data as well as minimizing re-partitions of intermediate data using coarse-grained data model and type-aware operators. Such benefits of the type-based approach is also likely to solve issues arise while processing UCQs. However, this approach is developed for basic graph pattern matching queries without considering any unions or disjunctive predicates and therefore we discuss how we advance the approach for optimizing UCQs in the next section.

5.3 Optimizing UCQs Using Type-based Semantic Reformulation

One possible way to evaluate UCQs using R-Type query processing model is to process UCQs using existing operators and unions, i.e., interpret patterns in union branches as independent basic graph patterns, so that each branch can be processed as a separate subquery and later merge the result using unions. However, this would likely to lead to similar issues in relational approaches such as resulting in a significant number of operators that need to be executed. Alternatively, as we alluded in Introduction, it would be reasonable to exploit a characteristics of UCQs, i.e., a large number of triple pattern across union branches are often common. The evaluation of such common triple patterns do not need to be repeated but can be shared as optimizations. To achieve a shared evaluation of common, we would need...
an extended matching process which ‘co-evaluate’ a star pattern with its all rewritings within in the single set of extended operators. This will also allow achieving other goals such as minimizing the number of operators needed while we still can enjoy the benefits of the R-Typing model such as early-pruning of irrelevant types without involving joins.

However, one major challenge to overcome is that the reformulated graph patterns are often heterogeneous subgraph patterns that consist of star patterns connected by chain joins. While applying the function \textit{REWRITE} over graph patterns generated by \textit{R3} and \textit{R4} eliminates all schema triple patterns and simplifies structures of reformulated graph patterns, a graph pattern resulted from \textit{R5} contains chain-joins, i.e., the resulting graph pattern contains the ‘inverse’ triple patterns whose objects are the subject of the original triple pattern which is connected via a chain-join. Fig. 5.4(b) shows that the original query \textit{Q} consists of a single star pattern with an \texttt{rdfs:type} triple pattern, but rewriting this query using \textit{R4} and \textit{D} leads to the graph pattern \textit{Q}' that consists of the two star patterns connected by a chain-join via the variable \texttt{?c}, e.g., \langle ?s ub:teacherOf ?c \rangle and \langle ?c ub:name 'Course0' \rangle. While these chain-joins could be processed using join operations, they will obviously involve additional data re-partitions across the cluster. Therefore, we need to develop another optimization technique that eliminate chain-joins, which is discussed in the next section.

5.3.1 Rewriting Techniques for Eliminating Chain-Joins via Inverting

One key observation that may be useful to solve this chain-join issue is that an inverse triple pattern is not connected with other triple patterns except its chain-join, i.e. the Subject variable of the inverse triple pattern is not overlapped with other variables in a query. Therefore, it may be possible to consider eliminating the chain-joins by ‘inverting’ the inverse triple patterns so that a chain-join can be transformed into a star-join. For example, \langle ?s ub:teacherOf ?c \rangle in \textit{Q}' can be inverted into \langle ?c ub:teacherOf ?s \rangle which transforms the chain-join into the star join as shown in Fig. 5.4(b). However, simply inverting triple patterns would not be sufficient due to a number of issues, e.g., (i) inverting triple patterns would result in changing semantics of queries and (ii) inverted triple patterns need to be differentiated with original triple patterns to avoid inadvertently adding superfluous triple patterns in queries.

To address these issues, we rewrite both query and data. Specifically, we introduce inverted triples matching inverse triple patterns and annotate inverse triple (patterns) with special markers to denote that they are inverse ones. The concept of the inverting can be formalized as follows.

\textbf{Definition 5.3.1.} (Inverting Triples) Let \( H \) be a set of schema triples and \( t_r \) be a triple of the form \( \langle s, p, o \rangle \). \( t_r \) is defined as range-invertable triple if \( (p, \texttt{rdfs:range}, c) \in H \). Inverting \( t_r \) forms a new inverted triple \( t_r^{-1} \) with the annotated Property \( p^{-1} \), i.e., \( \langle o, p^{-1}, s \rangle \).

\textbf{Example 5.3.1.} (Generating Inverse Triples) Fig. 5.4(a) shows that the triple \( \langle \texttt{ub:prof1}, \texttt{ub:teacher0f}, \texttt{ub:c1} \rangle \) is the range-invertable triple because the ontology \( O \) contains \texttt{rdfs:range} axiom for the Property \texttt{ub:teacher0f}. Thus we generate an inverse triple \( \langle \texttt{ub:c1}, \texttt{ub:teacher0f}^{-1}, \texttt{ub:prof1} \rangle \) and
5.3.2 Extending R-Type Query Processing Model for UCQs

Reformulation techniques presented in the previous section allow us eliminating all chain-joins in reformulated star patterns by rewriting queries and data, which provides opportunities to allow a star pattern and its reformulations to be co-evaluated using type-based query processing model. However, current R-Type operators are built on the assumption that each operation is responsible for processing a single star pattern in a query. In other words, each operator currently accepts a type signature of a single star pattern as input parameter and process types based on this single type signature (a single set of Properties) for selecting type names or projecting irrelevant triples. To co-evaluate multiple star patterns (i.e., the original star pattern and all of its reformulations) with these operators, we need the operators which can process type names and triplegroups based on multiple type signatures (rather than a single type signature) which are denoted as Multi-TypeSelection \(\pi^{\gamma+}\) and Multi-Projection \(\sigma^{\gamma+}\). Essentially \(\pi^{\gamma+}\) selects type names matching multiple type signatures and \(\sigma^{\gamma+}\) will project out irrelevant triples among typed triplegroups based on multiple type signatures. To feed these multiple type signatures, we need to extend notations and convenience functions as follows.

Let \(tp\) be a triple pattern and \(\{tp\}^+\) be a set of triple patterns generated by applying reformulation rules and REWRITE over the \(tp\). Let \(stp\) also be a star pattern that consists of triple patterns \(\{tp_1, tp_2, \ldots, tp_n\}\) and \(\{stp\}^+\) be a set of reformulated star patterns of \(stp\) which can be defined using Cartesian products of triple patterns as follows.

\[
\{stp\}^+ = \{\{tp_1\}^+ \times \{tp_2\}^+ \times \ldots \times \{tp_n\}^+\}.
\]

In other words, \(\{stp\}^+\) can be defined as a combination of all possible reformulations of star pattern \(stp\). Based on these notations, we extend convenience functions to extract type signatures of multiple star patterns so that operators can accept type signatures of multiple star patterns. Let us first override \(prop()\) for convenience so that \(prop(stp)\) returns distinct Properties in \(stp\). Let \(propset(\{stp\}^+)\) also be a function such that \(propset(\{stp\}^+)\) returns sets of Properties in reformulated star patterns, \(propset(\{stp\}^+) = \{prop(stp_1), prop(stp_2), \ldots, prop(stp_m)\}\). The interpretation of reformulated star patterns \(\{stp\}^+\) (or \([[\{stp\}^+]\]]) can then be expressed in terms of the existing operators with extended input parameters as follows.

\[
[[\{stp\}^+]] = \pi^{\gamma+}_{propset(\{stp\}^+)}(\sigma^{\gamma+}_{propset(\{stp\}^+)}(PN)) \quad (5.1)
\]

In other words, the interpretation of reformulated star patterns is almost similar to the interpretation of a star pattern except the use of extended operators for multi-selecting and multi-projections of type-
Figure 5.5: Interpreting ontological query $gp$ as (a) unions of FOL formulas and (b) disjunctions of MSO formulas.

matching triplegroups.

**Example 5.3.2.** (UCQ Processing Using R-Type Query Processing Model) Fig. 5.5(b) shows the execution plan for $gp$, which is expressed using the extended operators. $\text{propset(}\{\text{stp}_{\text{et}}\}\rightarrow)$ in the left subplan returns (i) the properties of $\text{stp}_{\text{et}}$, $\{\text{takesCourse}, \text{rdf:type}\}$, and (ii) the properties of the reformulated star patterns, e.g., $\{\text{takesCourse}, \text{studMemberOf}\}$ from $\text{stp}_{\text{es}}$. Applying $\sigma\gamma^+$ returns all type names relevant to these star patterns, e.g., $\tau_{\text{et}}, \tau_{\text{est}}, \tau_{\text{estp}}$, etc. $\pi\gamma^+$ then selects triplegroups of relevant types and eliminate irrelevant triples in these triplegroups, returning triplegroups that exactly match $\text{stp}_{\text{et}}$ and its reformulations. The right subplan also can be similarly processed as the left one. Note that (i) this plan only consists of 5 operations and requires 1 data repartition for joining relevant triplegroups and (ii) the number of operations and the plan structure are not affected by the complexities of ontologies. On the contrary, processing the initial UCQs built on relational algebra in Fig. 5.5(a) involves 400 selections and 300 joins that lead to 300 repartitions of data. The number of operations would also increase if the number of relevant axioms would increase.

As a final note, range-invertable triples introduced by semantic optimizations do not produce any superfluous results which can be proved as follows.

**Lemma 5.3.1.** Introducing range-invertable triples does not cause any superfluous answers.

**Proof.** A set of Properties used for range-invertable triples are specially annotated, therefore they are fully disjoint with a set of original, non-inverse Properties in datasets. These specially annotated Properties are not exhibited to users. Consequently, such Properties cannot be used for writing queries hence no spurious results are introduced. 

While inverse Properties cannot be directly used for writing queries, they are used for eliminating chain joins generated by $\text{rdfs:range}$ axioms during processing queries.

**Example 5.3.3.** (Query Processing over Triplegroups with Inverse Triples) Fig. 5.4(b) illustrates that the triple pattern $\langle ?c, \text{rdf:type}, \text{ub:Course}\rangle$ in Q2 is first reformulated into $\langle ?x, \text{ub:teacher0f}, ?c\rangle$.
Table 5.1: Dataset characteristics

<table>
<thead>
<tr>
<th></th>
<th>Size (GB)</th>
<th>#Triples</th>
<th>#Properties</th>
<th>#Classes</th>
<th>#Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM 1k</td>
<td>23.8</td>
<td>138M</td>
<td>18</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>LUBM 2k</td>
<td>47.9</td>
<td>276M</td>
<td>18</td>
<td>15</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 5.2: Storage requirement and preprocessing time of the benchmarked approaches

<table>
<thead>
<tr>
<th></th>
<th>Storage requirement (GB)</th>
<th>Pre-processing time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUBM 1k</td>
<td>LUBM 2k</td>
</tr>
<tr>
<td>Raw</td>
<td>23.8</td>
<td>47.9</td>
</tr>
<tr>
<td>Hive (U, N, M)</td>
<td>13.7</td>
<td>27.7</td>
</tr>
<tr>
<td>Hive (D)</td>
<td>17.5</td>
<td>35.2</td>
</tr>
<tr>
<td>SemStorm+</td>
<td>10.2</td>
<td>20.6</td>
</tr>
</tbody>
</table>

in Q2′ using rdfs:range axioms. The chain join in Q2′ is rewritten into a star join in Q2'' by inverting (/?x, ub:teacherOf, ?c) into (?c, ub:teacherOf−1, ?x). This reformulation allows triplegroups with the types τ_{t-1}\ldots to be matches for Q2'', e.g., a triplegroup \(tg_2\) in Fig. 5.4(a) have the identical graph structures and Properties, which can be used as answers for Q2''.

5.4 Empirical Evaluation

5.4.1 Testbed Setup

The goal of our evaluation was to compare the performance of our approach with other rewriting techniques based on relational algebra.

**Benchmark Dataset and Queries.** Our testbed consists of *Lehigh University Benchmark* (LUBM) [38] benchmark datasets, which are synthesized based on an ontology from academic domain. Testbed queries were adapted from LUBM benchmark queries to incorporate scenarios on varying characteristics of UCQs such as sharing the evaluation of common expressions and the impact of semantic optimizations. Query selectivities were also varied by binding Object fields of triple patterns (high-selectivity, denoted with postfix a), and evaluating same query with unbound Object (low selectivity marked with b). We observe that the execution of some queries required significantly longer time than others, thus we set a timeout period, i.e., we aborted the execution of queries after the specified time period has passed (e.g., 1 hrs was set as timeout for LUBM 1k) and marked such queries using a blue-colored tilde. Approaches that failed to produce answers due to errors such as job failures, are marked using a red-colored ×. Additional results with real-world datasets are available on the project site [2].

**Evaluated Approaches.** We extended our previous type-based query processing system to include query rewriting techniques and semantic optimizations presented in this paper and we denoted the ex-
tended system as SemStorm+. To evaluate other types of query rewritings, we considered employing recent Hadoop-based RDF query processing systems [44, 34, 71, 63, 54, 16, 86]. However, all of them do not fully support ‘all’ constructs which are necessary for rewriting queries such as FILTER, OPTIONAL, and UNION. Therefore, we employed a recent stable version of Apache Hive (1.2.2) for simulating four types of query rewriting techniques we covered in Section 5.2. Note that Hive does not use SPARQL as query languages, thus we translated SPARQL queries into equivalent form of HiveQL queries, which have almost the same syntax as SQL.

1. **Hive(U)** simulated a default query rewriting that generates UCQs (i.e., the hybrid approach discussed in Section 2).

2. **Hive(M)** rewrote UCQs using MQO, i.e. non-overlapping patterns in union branches are mapped into OPTIONAL clauses. Because HiveQL does not support OPTIONAL constructs, we use left outer joins to simulate the semantics of OPTIONAL constructs. We additionally inserted a post-processing operation that (i) extracts triples matching each subqueries (or subpatterns in union branches) from the result of left outer joins and (ii) merge them using unions.

3. **Hive(N)** reformulated UCQs using union pushdowns, i.e. triple patterns and their rewritings are nested (grouped as subqueries) and join with other triple patterns.

4. **Hive(D)** transformed UCQs as a basic graph pattern matching queries with disjunctive predicates and objects.

To simulated these approaches, RDF triples were loaded into HDFS and imported into a ternary table $T$. As optimizations, all approaches except the **Hive(D)** employed the vertical partitioning approach, which partitions table $T$ partitioned based on Property types using a map-only job per Property type. We also note that (i) the reformulation introduced in Section 5.3.1.1 (i.e., reformulating graph patterns to eliminate schema triple patterns) are also applied to all other approaches so that the number of union branches to be evaluated can be equal for all approaches and (ii) all approaches are executed using Tez [4] instead of MapReduce, so that we gave more opportunities for Hive to use more flexible and optimized query execution plans. Though our focus is to evaluate rewriting techniques in the distributed contexts, we additionally included RDF-3x for completeness, which is the state-of-the-art centralized RDF query processing system. Similar to Hive, we tested four types of rewritings for RDF-3x as well, but the result is not included due to a space constraint. All tested queries and other additional results are available at our project site [2].

**Cluster Configuration.** Evaluation was conducted on 45-node and 90-node Hadoop clusters in VCL\textsuperscript{2}, with each node equipped with Xeon dual core x86 CPU (2.33 GHz), 4GB RAM, and 40GB HDD. While cluster sizes may appear larger than necessary, the 45-node and 90-node cluster made available 900GB and 1.8TB disk space (20GB per node) which was required to store the intermediate

\textsuperscript{2}http://vcl.drupal.ncsu.edu
data materialization, e.g., pre-processing of the 450GB LUBM dataset materialized 2-3 times of the input size. For RDF-3x, a dedicated single physical node was used, with Xeon quad-core CPU (E5410, 2.33GHz), 40GB RAM, and 4TB HDD. All results were averaged over three or more trials.

5.4.2 Evaluation Results

Pre-processing Input Datasets

Table 5.1 shows the characteristics of datasets used in this evaluation and Table 5.2 presents the storage requirement of SemStorm+ and all other rewriting techniques using Hive. For all approaches, we shortened URIs with pre-defined prefixes used in LUBM (:ub), producing QNames for URIs, e.g., \(<\text{http://www.w3.org/1999/02/22-rdf-syntax-ns#type}\>\) is mapped into rdf:type. Hive(D) uses a ternary relation \(T\) without any additional preprocessing thus its preprocessing time was the shortest one but its storage requirement was much larger than others. Other Hive-based approaches used the VP approach thus its processing time was longer but required a less storage space due to the removal of repeated Properties. As shown in the previous chapter, SemStorm+ required a less amount of storage space compared to all Hive approaches because of its efficient storage model, e.g., common nodes across triples in triple-groups or triplegroups of the same type are only stored once without duplications. We also measured the requirement of space and execution time for semantic optimization in SemStorm, i.e. materializing types generated by inverting triples for eliminating chain-joins. Among 31 property types in LUBM datasets, 18 property types of LUBM has relevant \(rdfs\:\text{range}\) axioms, which takes up approx. 23\% of datasets. We note that this applied for all LUBM datasets regardless of their sizes because LUBM generally maintains the number of the Property types and the ratio of triples having such Property types equally. We observed that approx. 8\% and 5\% of additional space and time were required, which is a reasonable trade-off for eliminating chain-joins in UCQs. We also note that the space requirement decreased from 23\% to 8\% for inverse triples because of the efficiency of SemStorm+’s storage model.

We also note that we considered larger LUBM datasets for evaluating rewriting techniques such as LUBM 5k, 10k, or 20k. We indeed began with the evaluation using such datasets and soon found that most approaches except SemStorm+ cannot proceed tasks due to the lack of resources such as memories for most Hive-based approaches, thus we used relatively smaller datasets that can be used for running all approaches. Nonetheless, we observed some complex queries were failed or timeout. Additional results tested using larger datasets are also available on the project site [2].

Varying the Number of Union Branches

In this task, we varied the number of rewritings (union branches in UCQs) with the fixed size of common subexpressions in UCQs and studied its impact. We used the LUBM ontology to build a custom query set as follows. Query A consists of a single triple pattern with Property rdf:type and Object ub:Person.
We then adjusted the number of axioms so that the query can be expanded from 0 to 5 union branches (A1 to A6). Query B and C are the same as A but additionally include a single non-expandable triple pattern that shares the same subject variable but had different selectivities. Query D included the second expandable triple pattern with Property rdf:type and Object ub:Publication which was chain-joined with the triple pattern in A via a single non-expandable triple pattern. We fixed the number of union branches for ub:Publication (8 branches) and increased the number of branches for ub:Person from 1 to 4, generating four queries (D1-D4) that have 8, 16, 24, and 32 of branches. Query E was similarly constructed but it included additional non-expandable triple patterns.

As the number of rewritings increased, we observed different trends for different rewriting techniques. Execution times of the tested queries are shown in Fig. 5.6(a-b).

1. **Hive(U)**: Hive translated UCQs into the plan that consists of groups of selections and joins where the number of groups were equal to the number of union branches in UCQs. Hive then executed operations of each group and then stored its intermediate result in HDFS. These results were later then merged into files as output. Note that Hive did not execute these operations in a sequential manner but launched every group of operations together to maximize the parallelism. This approach worked in a small number of union branches, but its execution time rapidly increased as the number of branches increased (e.g., #branches increased from 8 to 32 in E1-E4). This is because adding union branches led to adding a number of operators such as tablescans and joins in the plans, causing overhead to (i) the application master node that manages the execution of query plan and (ii) the slave nodes.
that executes actual queries. In other words, a large number of operators were concurrently executed while the number of execution slots was limited, thus the execution of operators were extremely slow. For example, the plan for \( E4 \) consisted of approx. 500 types of operators. For these reasons, execution times of queries often were longer than other approaches, causing timeouts (\( E1-E4 \)). Hive also sometimes cannot process plans and failed to process queries such as \( D1-D4 \) even if the syntax of queries were correct, which shows that the execution of UCQs in existing query processing framework is indeed challenging. We also note that similar issues were observed in relational databases as well [25].

2. **Hive(N)**: Unlike the previous approach, the overall execution flow of this approach is first scanning and annotating (i) all relations for expanded triple patterns and (ii) relations for non-expanded triple pattern in the map phase. Union is implicitly processed, i.e. all different triples matching expanded triple patterns are merged via the shuffle and merging process between map and reduce phases. These tuples are then sort-merge-joined in reduce phase. Therefore, adding new union branches only add additional tablescan operators without adding any join operators, which led to more amenable execution plans and therefore in many cases this approach was better than \( \text{Hive(U)} \). However, its performance was not far better because, like \( \text{Hive(U)} \), this approach also had to transfer all triples matching union branches to reducers as a part of implicitly processing unions, while many of transferred tuples would likely to be irrelevant ones which are later filtered out using joins in reduce phase. The number of such triples increased fast in queries such as \( D1-D4 \) and \( E1-E4 \) and therefore the execution time significantly increased.

3. **Hive(D)**: Hive did not apply for any OR expansions for disjunctive queries built for \( \text{Hive(D)} \), thus this approach processed queries as we planned, i.e. evaluating without any unions but selections with disjunctive conditions. This allowed a shared the evaluation of reformulated triple patterns by scanning the triples only once. For this reason, compared to the other approaches, its execution time was slowly increased and stable. However, as the number of filter conditions (union branches) increased, its execution time also became longer because a single triple needs to be evaluated over many filter conditions, delaying scanning process. The trend became worse as the number of triple patterns increased because this approach cannot utilize vertical partitioning but had to scan a whole relation \( T \) several times for every triple pattern in queries (e.g., \( E1-E4 \)).

4. **Hive(M)**: This approach showed better performance than the previous approaches for a small number of rewritings because it shared the evaluation of common subexpressions. However, we observed it became significantly slower as the number of branches increased because the size of intermediate results rapidly increased as the number of left outer joins increased. We observed that this significantly impacted the performance in distributed environments because a huge amount of intermediate data needs to be shuffled and sorted for join operations. This issue was particularly severe when the number of branches (optional clauses) increased fast such as \( E1-E4 \), causing failures during query
Figure 5.7: Execution time of queries with a varying size of common expressions: (a) high and (b) low selectivities (unit: s)

Figure 5.8: HDFS read of queries with a varying size of common expressions: (a) high and (b) low selectivities (unit: GB)

processing due to the OOM.

5. **SemStorm+**: As we increased the number of branches, the execution time of *SemStorm+* was also increased. However, its increasing rate was extremely low because the burden of processing for each branch was relatively smaller compared to other approaches. In other words, unlike other approaches, adding union branches only led to the increment of matching types using a fixed number of operations. Therefore, even for complex queries such as *E1-E4*, the performance of *SemStorm+* was relatively stable and much faster than others.
Varying the Size of Common Subexpressions

This task studied the impact of the size of common subexpressions in queries, i.e. the number of triple patterns which are not expanded due to the absence of related axioms in ontologies. Our assumption was that such non-expanding triple patterns would be repeatedly evaluated in other approaches such as Hive(U), while SemStorm could share the evaluation of such patterns. To test this assumption, we built a custom query set based on LUBM datasets where query \( F1 \) consists of a single triple pattern with Property \( \text{rdf:} \text{type} \) and Object \( \text{ub:} \text{Person} \) which is expanded into 14 different triple patterns while other queries are single star pattern queries that include additional non-expanding triple patterns. Queries \( F2-F4 \) include the single non-expanding triple pattern with three different Properties that show decreasing selectivity (increasing popularity). Similarly, query \( F5-F7 \) included the two non-expanding triple patterns which use the same set of properties in \( F2-F4 \). Similarly, \( F8 \) contains three non-expanding triple patterns. We also tested the assumption using other class in ontology, i.e. query \( G1 \) consists of a single triple pattern with Property \( \text{rdf:} \text{type} \) and Object \( \text{ub:} \text{Publication} \) which produces 8 different triple patterns. We similarly designed the query \( G2-G3 \) and \( G4 \) to include the single and the double non-expanding triple patterns. Query \( H1 \) is the two star pattern query built by combining \( F1 \) and \( G1 \). \( H2 \) is similarly built but additional non-expanding triple patterns are included for each star pattern. Execution times for different approaches are shown in Fig. 5.7(a-b).

We analyze each approach and its result as follows.
1. Hive(U): We observe that Hive did not specifically transform initial query plans to share the evaluation of non-expanding triple patterns, thus the same vertical partitions (relations) were repeatedly processed for non-expanding triple patterns. This execution strategy therefore led to the excessive uses of task slots and I/Os at the same time, slowing down overall processing performance. This issue was particularly severe in the case where (i) the number of branches are relatively high and (ii) The selectivities of non-expanding triple patterns are low and joins are involved over such triple patterns because several groups of maps and reduces that use a large number of tasks were concurrently executed. These issues arose particularly for queries including a triple pattern \( \text{ub:} \text{Person} \) such as queries \( F2,F5,F6 \) and \( H1 \).

Our initial expectation was that Hive(U) would be the slowest one in this scenario but was relatively faster than our initial expectation. We observed that, if the number of joins became large, Hive often used the cost-based optimization such as using broadcast map-side joins for small or empty relations to avoid that every branch is mapped to expensive aggregation/reduce phases. For example, \( F2 \) initially required 28 operators for two triple patterns in 14 union branches but some mappers were eliminated using the cost-based optimization and eventually 22 operators were processed. While this optimization could relieve the burden of processing of some branches, it was not sufficient once the number of branches in queries becomes very large such as \( H1 \) and \( H2 \). As discussed in the previous task, too many operators were generated in the end, causing timeouts or failures due to the OOM.
2. **Hive(N)**: compared to the performance of Hive(U), this approach was able to shorten the execution time by sharing the evaluation of common subexpressions and it was particularly effective when the selectivities of common subexpressions were low such as $F_2$, $F_5$, and $F_6$. However, this approach was not very effective if the selectivities of common subexpression were relatively high such as $F_3$ and $F_4$ because the benefits of sharing the evaluation was not a significant.

3. **Hive(D)**: This approach were able to share the evaluations of common expressions across different union branches as disjunctive predicates, thus overall performance was better than other Hive-based approaches. However, similar to the previous task, the relation $T$ had to be repeatedly scanned for each triple pattern. Therefore, a large amount of irrelevant tuples were inevitably transferred across nodes when joins are involved, causing timeouts for complex queries such as $H_1$ and $H_2$.

4. **Hive(M)**: This approach overall showed comparable performance by sharing the evaluation of common expressions, but the execution time becomes substantial when the selectivities of common expressions are low such as $F_2$ and $G_2$, where all the `ub:Course` and `ub:Publication` match common subexpressions respectively. In such queries, the size of intermediate data became extremely large because of repeated left outer joins over large intermediate data matching common subexpressions, involving a substantial disk and network I/Os and memories. In the end, this approach also cannot process complex queries involving a large number of optional clauses such as $H_1$ and $H_2$, resulting in failures.

5. **SemStorm+**: Similar to the previous task, SemStorm showed the best performance for all queries. The gaps between other approaches and ours were particularly larger for queries with low-selective common subexpressions because ours can effectively share the evaluation of common subexpressions such as queries $F_2$, $F_6$, and all $H$ queries. To additionally measure the effectiveness of sharing scans in SemStorm, we disabled the sharing of scans common triples in triplegroups between types, so that SemStorm+ had to processing types again. The comparison of these results are shown as $SemStorm+(On)$ and $SemStorm+(Off)$ in Fig. 5.7. Overall, compared to $SemStorm+(Off)$, $SemStorm+(On)$ showed from 2% up to 25% performance enhancement and it was particularly effective for queries with low-selectivity common patterns in queries such as $F_2$ and $F_6$.

We additionally measured the HDFS disk read for these queries and its results are shown in Fig. 5.8. Generally, $SemStorm+$, $Hive(N)$, and $Hive(M)$ required a smaller amount of HDFS read because those approaches shared the evaluation of common subexpressions. $SemStorm+(On)$ particularly used up a minimum amount of disk I/Os for queries with low-selectivity queries. $Hive(D)$ also shared the evaluation but the amount of HDFS read was very large because it cannot use VP.

### Varying the Number of rdfs:range Axioms

This task studied the effectiveness of the semantic optimizations that eliminate chain-joins in UCQs. For this study, this time we disabled the reformulation in the implementation of $SemStorm+$ and measured its
impact. The SemStorm+ with reformulations and without reformulations were denoted as SemStorm+ (wS) and SemStorm+ (woS), respectively. We then built a set of queries for this task. A query set Js were constructed from J0, a star pattern query that consists of two triple patterns: the first triple pattern (tp1) includes Property rdfs:type and Object ub:Course, which was later expanded into 4 different triple patterns, and (ii) the second triple pattern (tp2) contains Property rdfs:type with bounded/unbound Object. We also adjusted LUBM’s ontology so that all Js queries only have two branches: the branch that contains J0 and the branch that includes a reformulation of J0 using rdfs:range axioms for ub:Course. The second branch therefore contained a single chain-join. J1 had the reformulated triple pattern the with lowest selectivities while J3 had the one with the highest selectivities. We also compared with Hive(U) to see the performance of these queries using default rewritings. Fig. 5.9(a-b) shows the execution time of these queries. The overall trend was that our semantic optimizations were particularly effective for queries with low selectivities, i.e., the performance gaps between SemStorm+ (wS) and SemStorm+ (woS) was the largest for J1.

5.5 Chapter Summary

In this chapter, we discuss the issues and challenges in optimizing UCQs on distributed platforms. We study possible rewriting techniques for optimizing the execution of UCQs in the context of distributed platforms and overview their limitations. We also present rewriting techniques that simplify graph patterns and co-processing techniques based on R-Type query processing model to minimize the burden from repartitioning of intermediate data. These techniques are also fully integrated with a semantic optimization technique that eliminates chain-joins in UCQs which further optimizes the execution of UCQs in a distributed context.
Chapter 6

Conclusion and Future Work

6.1 Overview of Dissertation

This dissertation studied semantics-oriented optimization techniques for large semantic network processing in the context of distributed platforms such as Hadoop.

• In Chapter 1, we present a motivational example to highlight the challenges in processing and optimizing two common types of queries over large semantic networks: basic graph graph pattern matching and ontological queries. While extensive studies have been conducted on optimizing such queries, these queries often tend to be complex and the size and the complexity of networks are rapidly increasing more than ever, which results in taking a significant amount of time to produce answers or simply failing in the end. We identify such challenges are stemmed from the fact that most query processing and storage models are semantic-oblivious, which do not fully exploiting semantic properties of the RDF data model. We therefore focus on developing semantic optimization techniques which exploit semantic properties of RDF queries and data for optimizing query processing over semantic networks.

• In Chapter 2, we overview RDF(S), SPARQL, and MapReduce framework. We also cover SPARQL query processing using relational algebra in the context of MapReduce and its limitations. We summarize alternative data model and algebra called Nested TripleGroup Data Model and Algebra (NTGA) in the context of MapReduce.

• In Chapter 3, we focus on optimizing conjunctive graph pattern matching queries which contain semantically related subexpressions across multiple star patterns. Existing approaches often represent the subexpressions based on the first-order relational expressions and evaluate the expressions independently, which results in significant performance overhead. We reformulate such query expressions as semantically equivalent second-order expressions using NTGA, which allows evaluating all star subgraph patterns concurrently thereby processing all semantically related subexpressions in the pat-
terns using a single disjunctive operator. However, a triplegroup can potentially match multiple star patterns, thus ambiguities naturally occur during the matching process. We therefore extend NTGA operators to deal with such ambiguities and achieve the shared execution of semantically related subexpressions.

- In Chapter 4, we present automatic typing scheme which uses the complete description of resources as its logical types. This scheme automatically induces and assigns logical types of triplegroups, which allows for a selective retrieval of matching triplegroups based on types. Further, the scheme enables type-based semantic optimizations that leverage ontological axioms to identify and dematerialize ‘redundant’ membership assertions, which effectively reduces the footprint of physical models. The proposed optimizations are implemented on Hadoop to build a type-aware storage model called SemStorm which provide an efficient representation of typed triplegroups using column-oriented file format.

- In Chapter 5, we study optimization and execution strategies of UCQs, which are generated by rewriting conjunctive queries with respect to ontological constraints. Processing UCQs in distributed context often requires a significant number of operators and the repartitions of intermediate data, therefore we explore rewriting strategies of UCQs to optimize the execution using different types of techniques such as using union-pushdown and multi-query optimization techniques. However, we observe that such strategies were also often suffered from the inability of sharing the evaluation of common expressions or a large amount of intermediate data which are not pruned early but later pruned using expensive joins. To solve such issues, we extend the type-aware query processing model to share the evaluation of contents-equivalent disjunctive subexpressions in UCQs as well as pruning out irrelevant tuples as early as possible by pruning out irrelevant types. This component also studies semantic optimization techniques that exploit the characteristics of rewriting rules for generating UCQs, such as pre-materializing types induced from rewriting rules and ontological axioms to achieve chain-join eliminations and simplifies graph patterns.

At the end of each chapter, we also present evaluation results on synthetic and real-world benchmark datasets to confirm the effectiveness of our semantics-oriented optimizations such as reducing execution time and the I/Os involved while evaluating basic graph pattern matching queries and ontological queries on distributed platforms such as Hadoop.

### 6.2 Future Directions

In the future, we would like to further explore the possibilities of semantic optimizations that use other combinations of ontological axioms from more complex ontological languages such as OWL2 QL. Specifically, we would like to investigate such optimization techniques in the context of type-based query processing models to further reduces or share scans, computations, and intermediate data refer-
ences across basic graph matching queries and ontological queries. We so far only consider the case that a specific set of ontological axioms and rules are given with datasets. However, it is possible that some datasets are not accompanied with such axioms and rules and our techniques are only useful in a limited situation. Nonetheless, our techniques may be useful again if we explore the possibility of generating ‘synthetic’ ontological axioms and rulesets based on graph structures and labels which are commonly occur. We believe that such structures can be easily discovered and mined by applying existing graph mining techniques. We would also like to investigate more optimized physical storage models and optimization techniques for our type-based query processing model such as developing compression schemes for typed triplegroups. Finally, we would also like to investigate the problem of extending our approach for more dynamic workloads that involve frequent updates and query processing over RDF data streams.
REFERENCES


