ABSTRACT

RAVINDRA, PADMASHREE. Optimizing RDF Analytical Queries on MapReduce. (Under the direction of Kemafor Anyanwu.)

The broadened use of Semantic Web technologies to enable data integration solutions across domains has increased the amount of semi-structured data on the Web represented using the Resource Description Framework (RDF). This has led to a growing interest to support analytics over semantically integrated RDF data warehouses, such as analysis of patient and drug profile data in life sciences for more effective clinical trial recruiting, and analysis of people-places data by e-government decision makers to improve citizen facilitation. In order to support large scale RDF analytics, it is crucial to investigate how to leverage parallel processing systems such as MapReduce and extended systems such as Apache Hadoop, Pig, and Hive.

An RDF analytical query consists of, (i) *graph pattern matching* to compute query-relevant subgraphs, (ii) *grouping* desired attributes, and (iii) *aggregating* values. In general, graph pattern matching translates to several join operations due to the fine-grained nature of the RDF data model. Many analytical queries involve multiple groupings and aggregations over slightly different graph patterns, further increasing the number of join operations. Evaluating such join-intensive workloads on existing platforms results in lengthy execution workflows. The challenge with such lengthy workflows is the I/O and network transfer costs due to the intermediate data produced across multiple map-reduce phases. Such costs can be significant for workloads that produce large intermediate results. Consequently, it is critical to develop techniques that enable more nimble execution of such workflows.

In this dissertation, we present a holistic approach to minimize the I/O and network transfer costs while processing RDF analytical queries on MapReduce. Given that RDF analytical queries often involve repeated computations over slightly different graph patterns, query plans that enable shared execution of common subpatterns are likely to compile into efficient MapReduce execution plans. To achieve this, we propose the following three optimization techniques that exploit sharing opportunities while evaluating RDF analytical queries.

- First, we propose an algebraic optimization approach that enables shared execution of overlapping graph patterns, thus eliminating the multiple phases of I/O and materialization involved in evaluation of multiple graph patterns in an RDF analytical query. A decoupling of the grouping and aggregation definitions is used to enable ***sharing of scans and computations across the graph pattern matching phases, as well as the grouping-aggregation phases***. Such a rewriting results in an aggressive reduction in the length of execution workflows.

- Second, we propose an algebraic optimization of basic graph pattern queries using an alternative data model and algebra called the *Nested TripleGroup Data Model and Algebra* (NTGA). The NTGA query plans enable concurrent computation of star-shaped join subpatterns in a query, as opposed to existing systems that require a separate map-reduce cycle for each star subpattern. Thus, by enabling ***sharing of scans and computations across multiple star subpatterns***, the NTGA query plans result in reduced numbers of map-reduce cycles.

- Third, we propose strategies for efficient management of intermediate results while evaluating graph pattern queries with multi-valued and unbound properties. For such queries, normalized representations of intermediate results using relational join operations introduce redundancies in results. To mitigate the effects of such redundancies, the NTGA's nested data model and lazy *unnesting* strategies enable ***sharing of data references, scans, and computations***, thus reducing the footprint of intermediate results.

We extended Apache Pig's computational infrastructure to integrate the NTGA-based data model and operators along with the optimization strategies. Empirical evaluation on real-world and synthetic benchmark datasets demonstrate that the NTGA-based query plans result in shortened execution workflows with reduced footprint of intermediate results, thus minimizing the I/O and network transfers while processing RDF analytical queries on MapReduce.

Optimizing RDF Analytical Queries on MapReduce

by
Padmashree Ravindra

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

2014

APPROVED BY:

Rada Chirkova	Munindar P. Singh
Nagiza Samatova	Kemafor Anyanwu Chair of Advisory Committee

DEDICATION

To my parents and grandparents.

BIOGRAPHY

Padmashree Ravindra was born to a loving couple, Ravindra Karkala and Latha Ravindra. She received her Bachelor of Engineering degree (Information Science and Engineering) in 2006 from Visvesvaraya Technological University (VTU), Bangalore, India. After working as a software engineer at Tata Consultancy Services Ltd. for two years, she chose to continue her graduate studies and received her Master of Science degree (Computer Science) in 2011 from North Carolina State University. Her research interests include optimization techniques for scalable processing of Semantic Web data and complex dataflows on cloud platforms. During her PhD program, she had internship experiences from IBM Research, India (Summer 2011) and IBM T.J.Watson Research Center, USA (Summer 2013).

ACKNOWLEDGEMENTS

The successful completion of this dissertation would not have been possible without the support and guidance of several people.

First and foremost, I would like to thank my advisor, Kemafor Anyanwu, who has not only helped me develop my research abilities through these years, but also provided support and encouragement during the rough and tough periods of my PhD program. During my initial years as an MS student, her encouragement to collaborate with a senior student allowed me to get early exposure to high-quality research, including paper writing and conference presentation. She was the first to identify my potential and mentored me during my transition from Masters to the PhD program. Throughout my PhD program, she has always encouraged me to apply for travel scholarships that allowed me to participate and learn from presentations at well established workshops and conferences. Her insistence to involve students in proposal writing has helped me broaden my research knowledge and understand the importance of well-motivated and high-quality proposals. Despite being a mother of three children and having family commitments, she has always been available to her students for guidance and feedback, sometimes even sacrificing her family time during weekends and special occasions such as birthdays. I am grateful for her valuable advice and feedback during submissions and presentations over the past six years, and am certain that this knowledge will continue to assist me during the rest of my research career.

I thank Rada Chirkova, Nagiza Samatova, and Munindar P. Singh for serving on my doctoral committee and providing valuable feedback on my dissertation research. I would also like to thank my academic grandfather, Professor Amit Sheth for encouraging me to pursue doctoral studies. I am grateful to my internship mentors, Rajeev Gupta (IBM Research, India) and Achille Fokoue (IBM T.J.Watson Research Center, USA) for their continued support and guidance that has extended beyond summer internships. I would also like to thank Ullas Nambiar (EMC, India) for his mentoring and constant encouragement throughout my PhD program. I 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 securing large scale Hadoop cluster reservations for evaluation purposes. 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) – Pradeep Murukannaiah, Radhika Sridhar, Vikas Deshpande, Haizhou Fu, Sidan Gao, Seokyong Hong, and HyeongSik Kim. I would like to specially thank HyeongSik Kim for his research collaborations and insightful discussions. This acknowledgment section would be incomplete without mentioning the numerous conversations

and discussions with my dearest academic sister, Sidan Gao. Our interactions have always helped me to be patient and positive during my PhD.

My stay at Raleigh would have been unimaginably boring without the love and support from my dear Sai family – Vijaya and Sudhakar Nori, Sri Vidhya and Sridhar Ramabadran, Rekha and Krishna Kumar, and all the Young Adult friends from the Raleigh Sai Center. I would like to specially thank Kushal Sheth, Aparna Bhaskara, and Priya Raghunanan for being a part of my life and sharing their love for Swami. Also, thanks to Madhuri Marri and Kunal Taneja for all the fun coffee conversations during our time together at NCSU. Thanks also goes to my room mates, Lakshmi Ramachandran and Anwesha Das, for their co-operation during my stay at NCSU. Needless to say, my friends Rajni Roshan, R. N. Anitha, Shridevi Bhat, Priyanka C.P., and Keshav D. Murthy have been very supportive despite being miles away. I feel very fortunate to know them and treasure their friendship.

I would not have set out to pursue graduate studies without the unconditional support from my loving parents, Ravindra Karkala (Appa) and Latha Ravindra (Amma). I would like to thank my younger sister, Prithvishree Ravindra, for her constant emotional support throughout my PhD. I am also grateful to my paternal grandparents – K. Sachidanandaiah (Ajja) and Mohini Sachidanandaiah (Ajji), and my maternal grandparents – K. V. Govindan (Thatha) and C. V. Seethamma (Paati), for all the love and support. Both my grandfathers played an important role in making me understand the significance of graduate studies. I would also like to acknowledge the support of my Uncles, Aunts, and Cousins. Special thanks to Uncle I.K. Ramesh (Doddappa) for supporting my graduate studies. I would like to thank my parents-in-law – T. V. Ramana (Uncle) and T. Adilakshmi (Amma) for encouraging me to continue my graduate studies even after my wedding. Needless to say, my husband, Suresh Thummalapenta has a major role in the successful completion of my dissertation. I shall be ever grateful for his patience and unconditional support through the ups and downs of graduate school.

As it is said, not even a blade of grass moves without the will of the Almighty. I cannot begin to express my gratitude towards my sweet Lord Sai, who allowed me to experience the life of a PhD student, held my hand through the successes and failures that came along, and gently pushed me towards the finish line of this PhD journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The broadened use of Semantic Web technologies for data integration solutions [32, 34, 55, 85] in domains such as e-governance, bioinformatics, and life sciences, has increased the amount of semi-structured data on the Web represented using W3C's metadata standard called the *Resource Description Framework* (RDF) [53]. Initiatives such as the Linking Open Data community project have led to the growth in the number of accessible RDF triples from two billion in 2007 to thirty one billion RDF triples in 2011 [21]. This evolution has led to a growing interest to support analytics over semantically integrated RDF data warehouses, as can be seen in the following two real-world use cases:

Use Case 1. *As a part of drug-discovery for cancer treatment, researchers want to identify chemical compounds that inhibit at least two proteins in the MAPK signalling pathway* [31].

The MAPK signalling pathway is responsible for cell proliferation, differentiation, and death, and hence is widely studied to treat cancer. In order to block such disease-related biological pathways, a potential drug must inhibit multiple targets (proteins) in the pathway. Deriving relationships between compounds and pathways involves retrieving information from three different datasets – details about chemical compounds and their biological assessments (*PubChem* dataset), target proteins that the compounds interact with (*Uniprot* protein dataset), and information about related pathways (*KEGG* pathway dataset). After retrieving chemical compounds that interact with proteins in the MAPK signalling pathway, the results need to be grouped based on the compounds and only compounds that inhibit at least two proteins need to be selected.

Use Case 2. *Health care policy makers want a report on the total number of deaths and the number of clinical trials for Tuberculosis and HIV/AIDS in all countries* [104].

Such a query is important for health policy makers of developing countries to analyze the disparity between biomedical research and the disease burden, and make policy decisions that can increase life expectancy and improve the quality of life in their countries. While information

about clinical trials and effectiveness of treatment options is available in *ClinicalTrials.gov*, statistics about mortality and burden of diseases for different countries is available in the *Global Health Observatory* (GHO), published by the World Health Organization. Information about biomedical research (MEDLINE publications and other life science journals) is available in the *PubMed* dataset. In order to generate the required report, the results need to be grouped based on country and disease, followed by aggregations on the number of clinical trials and deaths due to the concerned disease in each country.

These use cases illustrate the need to support analytical queries over semantically integrated RDF data warehouses, also reflected by the recent inclusion of grouping-aggregation constructs [43] in SPARQL [74], the standard RDF query language. Moreover, each linked RDF dataset may be large in size, e.g., *PubMed* contains 800 million RDF statements, thus requiring scalable solutions for analytics over RDF data warehouses. Hence, it is crucial to investigate how to leverage parallel data processing systems such as MapReduce [37] and Dryad [48], and extended systems such as Apache Hadoop [20], Pig [1], and Hive [94].

Relational vs. RDF Data Warehouse. In order to understand the issues and challenges in supporting RDF analytical queries, we first review the basic differences in the models and organization of a relational and an RDF data warehouse. Consider an example e-commerce use case from the Berlin SPARQL Business Intelligence (BI) benchmark [2]:

Query Q. *Retrieve the average price and the number of product features for combinations of product and the vendor's country*

In the case of a relational data warehouse, data is organized as "*fact*" and "*dimension*" tables. A fact table contains detail facts and measurements that are interesting information to a business process. A dimension table contains supporting details about columns referenced in the fact table. In the example e-commerce scenario, a relational data warehouse may consist a central fact table *ProductOffer* in a star or snowflake schema, surrounded by dimension tables *Product* and *Vendor*:

ProductOffer(ID, ProductID, Price, ValidFrom, ValidTo, DeliveryDays, VendorID)
Product(ID, Label, Type, Feature)
Vendor(ID, Label, Country, Homepage)

Answering an online analytical processing (OLAP) query such as query Q, involves aggregating the measure attributes (e.g., Price, Feature) in a fact table that is grouped by a set of dimension attributes (e.g., Product, Country).

On the contrary, RDF models data as 3-tuples or *triples* of the form (*Subject*, *Property*, *Object*) where a property defines a binary relationship between resources, or between resources and its attributes. For example, the triple ($\&Offer1$, *vendor*, $\&V1$) states that the subject resource $\&Offer1$ has a vendor (property) relationship with the object resource $\&V1$. Figure

Subject	Property	Object
&V1	type	VENDOR
&V1	label	Vendor1
&V1	country	US
&V1	homepage	www.vendor....
&Offer1	type	OFFER
&Offer1	vendor	&V1
&Offer1	product	&Prod1
&Offer1	price	108
&Offer1	delDays	2
&Prod1	type	PRODUCT
&Prod1	prodFeature	&Pf1
&Prod1	prodFeature	&Pf1
...	...	...

(a)

```
SELECT ?prod, ?country, AVG(?price),
COUNT(?prodFeature),...
{
  SELECT ?prod, ?country ...
  WHERE {
    ?vend homepage ?hpage .
    ?vend label ?vlabel.
    ?vend country ?country .
    ?offer vendor ?vendor .
    ?offer price ?price .
    ?offer delDays ?delDays .
    ?offer product ?prod .
    ?prod label ?pLabel .
    ?prod prodFeature ?pf . }
}
GROUP BY (?prod, ?country)
```

Basic Graph Pattern: SJ1, SJ2, SJ3; J1, J2

(b)

Figure 1.1: (a) Example RDF triple relation describing Product Offers, Vendor information, and Product details (b) Example RDF analytical query with grouping-aggregation over a graph pattern with three star subpatterns SJ1, SJ2, and SJ3

1.1(a) shows an example set of RDF triples describing Vendors, their product Offers, and Product details. A collection of RDF triples can also be modeled as a directed labeled graph with nodes representing subjects and objects, and labeled edges denoting the property types.

Given such a fine-grained data model in RDF, evaluating the example analytical query Q on RDF data consists of three components, (i) *graph pattern matching* to compute the query-relevant subgraphs corresponding to the product offers, features, and product vendors, (ii) *grouping* the resulting patterns based on the values of product and country combinations, and (iii) *aggregating* the values to compute the average price and the count of the product features. The graph pattern matching phase is typically join intensive and requires several join operations to reassemble the relevant "fact" and "dimension" information.

Additionally, relational data warehouses are usually centered around one fact table, while RDF data warehouses may involve multiple central concepts across semantically linked datasets that may be interesting to different users. Further, schema and relationships between entities are static in a relational data warehouse (fixed during design time). Such an assumption does not hold for RDF data warehouses where relationships between entities may evolve as new datasets are added. There is also a need to support flexible querying of RDF data to include scenarios where such relationships may be partially or completely unknown. Next, we review the basics of SPARQL query processing on RDF graphs.

Basics of SPARQL Query Processing. Consider the SPARQL query in Figure 1.1(b) that corresponds to our example query Q. The foundational construct for querying RDF data

is a *triple pattern*, which is a triple in which either of subject, property, or object components can be replaced by a variable (denoted by a leading '?'). For example, the triple pattern ($?s$ *vendor* $?o$) matches all triples with property *vendor*, whose subjects and objects are considered as variable bindings for $?s$ and $?o$, respectively. A relationship that is unknown or irrelevant (don't care edge) can be expressed using a variable in the property position of a triple pattern. For example, ($\&Offer1$ $?p$ $?o$) retrieves all triples with subject $\&Offer1$, with valid bindings for $?p$ and $?o$, respectively. Typical queries on an RDF database consist of a *Basic Graph Pattern*, which is a conjunction of two or more triple patterns. The basic graph pattern in our example query in Figure 1.1(b) consists of 9 triple patterns to retrieve details about product offers, along with their vendor details. The answer to a graph pattern consists of bindings that match all triple patterns.

RDF data is commonly stored as ternary relations and query evaluation is achieved using several relational style joins (shared variables across triple patterns denote equi-joins). Our example query can be expressed as a relational query with 8 self-join operations on typically large triple relations. Some systems use multi-indexing schemes [64, 100] for faster retrieval, while some others use the vertical-partitioning [6] storage model that enables joins on smaller property-based partitions of the triple relation. Another possible optimization [64, 98] is to partition the joins in the graph pattern based on the subject variable into *star subpatterns*. This is based on the observation that star subpatterns commonly occur in graph pattern queries, and can be evaluated as n-way relational joins (star-joins). Our example query in Figure 1.1(b) can be partitioned into three *star subpatterns* – SJ1, SJ2, and SJ3. Thus, relational-style processing of graph pattern queries typically consists of several such star-join structures and additional joins such as J1 and J2 in Figure 1.1(b), to connect the star patterns. Graph pattern matching is followed by evaluation of the grouping construct, which partitions the matched subgraphs into one or more groups (based on the values for product and month in our example). The aggregation function is then applied on each such group, to compute the average price and the count of features for products in each such group. In complex analytical queries requiring multiple grouping and aggregation phases, each such grouping may be associated with a slightly different graph pattern, further increasing the required number of join operations.

Scale-Out Processing. Given the significant costs of processing RDF analytical queries, it is crucial to investigate how the processing of such join-intensive workloads can be effectively supported on MapReduce-based parallel data processing systems. MapReduce's easy-to-use programming interface allows users to encode tasks as two primitive functions: *map* and *reduce*. Open-source implementations of MapReduce such as Hadoop [20] support fault-tolerance and automatic parallelization of MapReduce programs across a cluster of commodity-grade machines. Hadoop-based extended systems such as Apache Pig [70] and Apache Hive [94] support high-level languages that allow users to express declarative queries that are automatically com-

piled into logical, physical, and MapReduce execution plans. In the case of multi-join queries such as graph pattern queries, each join compiles into a physical operator that is executed in a separate map-reduce (MR) cycle. However, most existing systems [70, 94] can process n-way joins on equivalent join attributes (star-joins) in a single MR cycle. The example graph pattern in Figure 1.1(b) would require 3 MR cycles for the star-joins (MR_{SJ1}, MR_{SJ2}, and MR_{SJ3}) and 2 MR cycles to connect the star subpatterns (MR_{J1} and MR_{J2}). This is followed by an additional MR cycle MR_{G1} for the grouping and aggregation.

In reality, most analytical queries are often more complex. For example, given the previous graph pattern query, one may be interested to find '*for each product-country combination, count for each month of 2012 the number of offers whose price was between previous month's average price of product offers and following month's average price of product offers*'. Such a query requires computing aggregates on product offers outside the groups, i.e., offers from previous and following months. These results are then used to compute the required aggregation of count of offers. This is an example of multi-pass aggregation [26]. In the context of MapReduce-based processing, each such grouping-aggregation phase results in a separate MR cycle.

Each such MR cycle is associated with I/O, sorting, and network transfer costs, which compound across multiple cycles of a lengthy execution workflow. In general, the MR execution workflow for an RDF analytical query would be as follows:

$$\text{MR Workflow } W = <MR_{SJ1}, MR_{SJ2}, ..., MR_{J1}, ..., MR_{G1}, MR_{G2}, ..>$$

with MR cycles for computation of star-joins, followed by joins connecting the stars, and finally the required grouping and aggregations. Then, the overall processing cost of the workflow is:

$$\text{Cost}(W) = \text{cost}(MR_{SJ1})+\text{cost}(MR_{SJ2})+...+\text{cost}(MR_{J1})+...+\text{cost}(MR_{G1})+\text{cost}(MR_{G2})+..$$

Thus, lengthy workflows lead to performance inefficiency and an important optimization goal is to *minimize the length of the MR execution workflow* [10, 46, 102].

The intermediate data flowing through a MapReduce data processing workflow impacts the initial data reads in the map phase, the data shuffling (local disk writes, sorting, network transfer costs) between the map and reduce phases, and finally the cost of writing the output at the end of the reduce phase. Additionally, the total disk space required for the successful completion of a task in Hadoop is equal to the size of output for each of the intermediate MR cycles. Thus, large intermediate results impose a high demand on the required amount of disk space. Thus, another important optimization goal is to achieve *efficient management of intermediate results* while processing graph pattern queries on MapReduce.

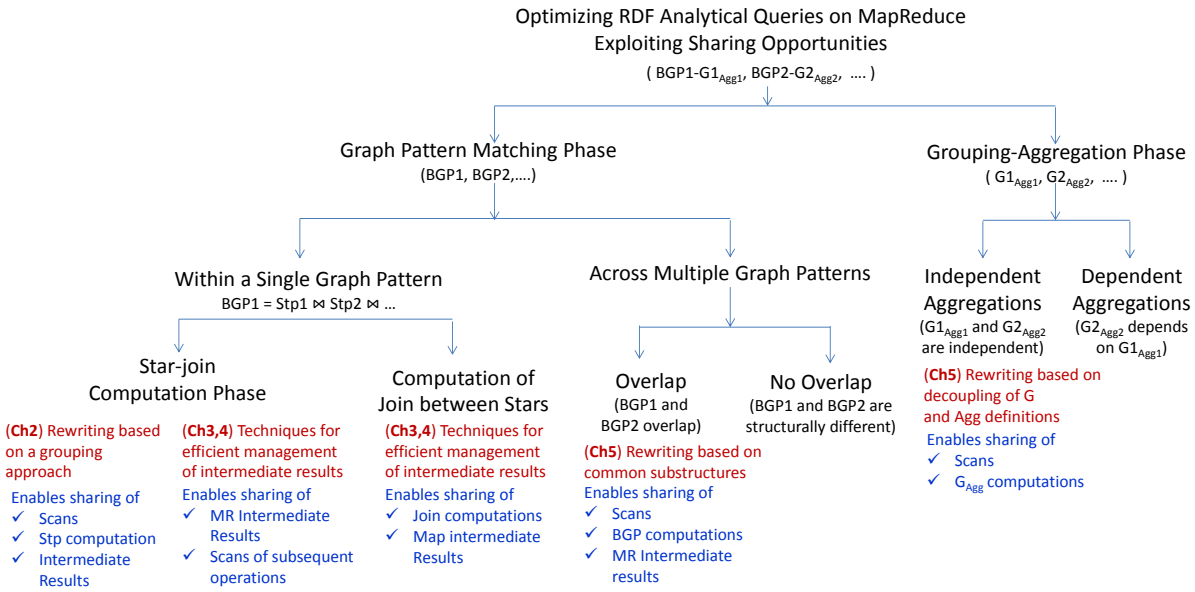


Figure 1.2: Exploiting sharing opportunities to optimize RDF analytical queries on MapReduce

1.1 Key Components

In order to minimize the I/O and network transfer costs involved in MapReduce-based processing of RDF analytical queries, it is essential to generate query plans that enable sharing of scans and computations during the graph pattern matching phases, as well as the grouping-aggregations phases. In this dissertation, we present optimization techniques that exploit such sharing opportunities to enable efficient processing of RDF analytical queries on MapReduce-based systems. First, we present an algebraic optimization technique that enables sharing of scans and join computations across multiple star subpatterns in a basic graph pattern query. Second, we present strategies for efficient management of intermediate results while processing graph pattern queries that introduce redundancy in intermediate results. These strategies enable sharing of intermediate data references, scans, and computations while evaluating graph patterns with multi-valued and unbound properties. Third, we present an algebraic optimization technique for more complex graph pattern queries with multiple grouping-aggregation constructs. The proposed techniques enable sharing of scans, computations, and intermediate data references across the graph pattern matching phases, as well as the grouping-aggregation phases. Figure 1.2 summarizes the different phases involved in evaluation of an RDF analytical query, the different optimization techniques presented in this dissertation, and the corresponding sharing opportunities that are enabled by each technique.

1.1.1 Algebraic Optimization of Basic Graph Pattern Queries to enable Sharing of Scans and Computations Across Star Patterns

This component addresses the problem of minimizing the number of MapReduce cycles while processing join-intensive workloads that are typical in RDF query processing. In this component, we consider basic graph pattern queries with bound property types.

Research Questions

Are there alternatives to the relational model and algebra that can be used to enable more efficient MapReduce query execution plans for RDF analytical queries? Assuming that such an algebra exists, are there different possible implementation strategies that could offer choices of execution plans given the requirements of a specific query? What foundations shall be used to establish equivalence between plans in this new algebra and traditional plans in relational algebra?

Proposed Solution

The motivation for this work stemmed from the observation [76] that any number of star subgraphs matching all star subpatterns in a query can be computed using a single MR cycle using a `GROUP BY` operation on the subject column of the triple relation. This eliminates the need for multiple MR cycles (one for each star subpattern) that are required to evaluate multiple star subpatterns using relational-style joins. In other words, given a query with n star subpatterns, the grouping-based approach results in an MR execution plan with n MR cycles, as opposed to ($2n$-1) cycles using the relational-style approach. However, the grouping-based star-join computation approach results in groups of triples or TripleGroups that are 'content-equivalent' to the n-tuples that result from relational joins. Hence, we require special TripleGroup-based operators and efficient MapReduce algorithms that enable efficient execution of RDF graph pattern queries. Specifically, this work makes the following contributions towards efficient evaluation of basic graph pattern queries:

- A TripleGroup data model and an algebra called the *Nested TripleGroup Data Model and Algebra* (NTGA) [77], that leads to efficient MapReduce execution plans with reduced numbers of MapReduce cycles.
- Operator implementation strategies that are integrated into our system *RAPID+* [14, 51] (NTGA-based extension of Apache Pig) to minimize costs involved in RDF graph processing.
- Establishing an equivalence between expressions in NTGA and relational algebra expressions representing a class of graph pattern queries. This equivalence is established by

introducing a notion of *content equivalence*.

- A comparative performance evaluation of RAPID+ using a benchmark dataset shows up to 60% performance improvement over the default Pig system for certain classes of graph pattern queries.

1.1.2 Evaluation Strategies for Efficient Management of Intermediate Results to enable Sharing of Intermediate Data References and Scans

Multi-valued properties that define one-to-many relationships such as Facebook friends and citation references are a common occurrence in real-world RDF datasets. While processing graph pattern queries involving such multi-valued properties using flat data models and their associated algebras, intermediate results contain redundancy due to repetition of single-valued attributes. Another such example is the case of graph pattern queries with 'unbound' properties (variable in the property position that represents don't care edges or unknown relationships) which are useful for flexible querying of RDF datasets. When evaluating unbound-property queries using relational-style operators on flat relations, the intermediate tuples contain redundant information related to the bound properties in the query. This is because the unbound-property column is repeated for each combination of the bound-property subtuple. Such redundancy in intermediate results have a tendency to aggravate with each subsequent join, thus negatively impacting the I/O, sorting, and network transfer costs of subsequent cycles. This component addresses the issue of mitigating the effects of redundancy in intermediate results while processing graph pattern queries.

Research Questions

What are the sources of redundancy in intermediate results while processing RDF graph pattern queries? What are the possible evaluation strategies that can be used to mitigate this redundancy in intermediate results while handling such queries?

Proposed Solution

NTGA's nested data model already enables concise representation of intermediate results. For example, m star subgraphs containing redundant information due to the presence of a multi-valued property with multiplicity m are implicitly represented in a single TripleGroup in NTGA. However, similar to systems such as Pig that support nested data models but do not support nesting-aware operators, the earlier generation NTGA operators require *unnesting* of such implicit representations prior to any subsequent join operation. This unnesting introduces redundancy in intermediate results. This work builds on the advantages of NTGA's nested data model and makes the following contributions to enable efficient management of intermediate results

containing redundancy:

- *Nesting-awareness* [78,81] in NTGA operators that allow delaying the unnesting of nested columns to a later phase in the MR workflow. This is in contrast to existing systems that support nested data models whose operators are not nesting-aware, and hence require that the nested column is unnested prior to any subsequent operation on the nested column.
- Logical and physical operators that enable *partial* and *lazy* unnesting strategies for joins on multi-valued properties [81] and for evaluation of unbound-property graph patterns [80]. These operators allow the expansion of intermediate results only when necessary, thus reducing the I/O, sorting, and network transfer costs to some extent.
- Extensive evaluation using large RDF graphs from both synthetic benchmark and real-world datasets demonstrates the efficiency of our approach over the relational-style processing of graph pattern queries with multi-valued and unbound properties in default Pig and Hive systems.

1.1.3 Algebraic Optimization of Complex RDF Analytical Queries to enable Sharing of Scans and Computations across Multi-aggregation Phases

Complex RDF analytical queries with multiple phases of grouping and aggregations result in complex evaluation plans [79] using relational-style operators. There are two major issues, (i) multi-pass aggregation queries result in redundant scans of the same relation even when the required aggregations are slightly different, and (ii) unlike the traditional OLAP systems which use optimized storage models (star or snowflake schema) for the fact and dimension tables, analytical queries on RDF data require additional joins to reassemble the fact and dimension information.

High-end OLAP servers (SAS, Teradata) rely on sophisticated indexing schemes and parallel architectures to combat the issue of redundant scans. Authors of *MD-Join* [26] show that decoupling the grouping and aggregation phases, not only enables more succinct expression of complex analytical queries, but also results in fewer scans of the fact tables. Prior work [89] explored the use of MD-Join for RDF analytical queries on MapReduce. However, direct application of such a decoupling has limited benefits in RDF analytics, due to the absence of optimized storage models such as fact and dimension tables. In this component, we present an algebraic optimization technique for RDF analytical queries that uses NTGA-based rewritings in the spirit of the *MD-Join* operator.

Research Questions

What are the optimization opportunities while evaluating analytical queries using NTGA? What

are the possible implementation strategies that can enable efficient MapReduce execution plans for MD-Join based evaluation of RDF analytical queries?

Proposed Solution

RDF analytical queries with multiple grouping-aggregation phases often involve overlapping graph patterns with common substructures. Such overlapping graph patterns can be expressed and evaluated as a *composite graph pattern*, that computes subgraphs matching all the original graph patterns. By doing so, we can share scans and computations across multiple graph patterns, which translates to a shortened MapReduce execution workflow with reduced I/O and network transfers. Further, such a rewriting based on the composite graph pattern, allows us to apply some generalizations of the *MD-Join* operator, such as parallel evaluation of independent aggregations. Specifically, this work makes the following contributions to enable efficient evaluation of complex RDF analytical queries using NTGA:

- Query rewrite rules to express overlapping graph patterns (in an RDF analytical query) as a *composite graph pattern*, based on common substructures. Decoupled reformulation of the grouping and aggregation definitions in the RDF analytical query expressed using the composite graph pattern.
- Logical and physical operators that enable efficient evaluation of a composite graph pattern. A logical and physical operator in the spirit of generalized *MD-Join*, that enables parallel evaluation of independent aggregations on composite graph patterns. These operators allow sharing of scans and computations across the graph pattern matching phase, as well as grouping-aggregation phases.
- Evaluation on synthetic and real-world datasets using analytical queries similar to those in the BSBM Business Intelligence benchmark, demonstrates the benefit of our approach over relational-style processing of RDF analytical queries in Hive. This holds true even for multi-query optimization based evaluation of RDF analytical queries on Hive.

1.2 Related Work

Some single-node RDF data processing approaches [64,98] apply sophisticated query optimization rules based on summary statistics collected during the pre-processing phase. RDF-3X [64] and Hexastore [100] use multi-indexing techniques that enable fast merge joins for join processing. All these approaches require heavy pre-processing, which may not be advantageous in the case of cloud based on-demand data processing tasks.

Optimizing MapReduce execution workflows. There have been different techniques proposed recently for optimizing MapReduce data processing workflows. While some efforts

[8, 10, 46, 77, 102] shorten the length of workflows to minimize the overall costs of MapReduce-based processing, some other efforts focus on sharing scans [68, 69, 99] and computations [68] across MapReduce workflows. The heuristic cost-based plan generator in [46] greedily groups the non-conflicting joins in a query to minimize the required number of MR cycles. HadoopDB [8] uses a hybrid database-Hadoop architecture to split execution between the database systems and Hadoop. However, the reduction in the number of MR cycles is limited to the portion of the query evaluation that can be pushed into the database, and is currently dependent on a tunable parameter [45] that determines the partitioning scheme. Techniques that follow replicated join scheme [10,102] to shorten the workflow, incur high data shuffling costs which is likely to worsen in the presence of multi-valued attributes. NTGA provides an algebraic optimization of graph pattern queries that is suitable for on-demand RDF querying systems that cannot rely on pre-processing. A more detailed discussion on different distributed query processing systems based on Hadoop and extended platforms is available here [15].

Efficient Management of Intermediate Results. In traditional database, the nested relational model [7] and object-oriented databases support sets, lists, maps and objects to concisely represent multi-valued attributes and complex objects with minimum data redundancy. Such extended models eliminate frequently occurring joins between objects by allowing nesting of related objects. Set-valued joins [44, 62] and other set-level comparisons [59] have been studied in the context of main-memory and centralized systems. The MapReduce-based approaches include work on sharing map output [68] across batch queries, re-using [38] parts of MR workflows in subsequent queries, and reduce routing strategies [63, 97].

Optimization of Complex OLAP queries. There has been a body of work to enable better expression and evaluation [12,25–29,40,41] of complex OLAP queries. While constructs such as the `CUBE BY` [41], grouping sets [25], etc., allow a user to have a finer control over the group specifications and aggregations, other works focus on efficient indexing [71, 101], materialized views [30, 42], and efficient evaluation of OLAP queries in distributed data warehouses [12, 13]. The authors of *MD-Join* [26] demonstrated that the decoupling of the grouping definition and aggregation computations not only allows more succinct expression of complex OLAP queries, but also enables better optimization opportunities. For example, the decoupled reformulation of complex OLAP queries using the *MD-Join* eliminates redundant scans and joins involving fact table for computing slightly different groupings and aggregations. In the context of MapReduce-based processing, an implication of such reductions in joins and scans of large fact tables is efficient execution plans with savings in I/O and network transfer costs. A more detailed discussion of benefits and issues in adopting such a decoupling of grouping-aggregation definitions for RDF graph analytics on MapReduce is included in Chapter 5.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 introduces the algebraic optimization of basic graph pattern queries using NTGA. Chapter 3 describes the proposed strategies for efficient management of intermediate results while processing graph pattern queries with multi-valued properties. Chapter 4 covers the extensions to NTGA to enable efficient evaluation of unbound-property queries. Chapter 5 introduces the algebraic optimization techniques for RDF analytical queries. Chapter 6 includes the conclusion and future work.

Chapter 2

Nested TripleGroup Data Model and Algebra

2.1 Motivation

With the recent surge in the amount of RDF data, there is an increasing need for scalable and cost-effective techniques to exploit this data in decision-making tasks. MapReduce-based processing platforms are becoming the *de facto* standard for large scale analytical tasks. MapReduce-based systems have been explored for scalable graph pattern matching [45, 46, 65, 66], reasoning [96], and indexing [47] of RDF graphs. In the MapReduce [37] programming model, users encode their tasks as *map* and *reduce* functions, which are executed in parallel on the Mappers and Reducers, respectively. This two-phase computational model is associated with an inherent communication and I/O overhead due to the data transfer between the Mappers and the Reducers. Hadoop based systems like Pig [70] and Hive [94] provide high-level query languages that improve usability and support automatic data-flow optimization similar to database systems. However, most of these systems are targeted at structured relational data processing workloads that require relatively fewer numbers of join operations as stated in [70].

On the contrary, processing RDF query patterns typically requires several join operations due to the fine-grained nature of RDF data model. Currently, Hadoop supports only *partition parallelism* in which a single operator executes on different partitions of data across the nodes. As a result, the existing Hadoop-based systems with the relational style join operators translate multi-join query plans into a linear execution plan with a sequence of multiple Map-Reduce (MR) cycles. This significantly increases the overall communication and I/O overhead involved in RDF graph processing on MapReduce platforms. Existing work [64, 98] directed at uniprocessor architectures exploit the fact that joins presented in RDF graph pattern queries are often organized into star patterns. In this context, they prefer bushy query execution plans over linear

ones for query processing. However, supporting bushy query execution plans in Hadoop based systems would require significant modification to the task scheduling infrastructure.

2.1.1 Related Work

Data Models and High-Level Languages for cluster-based environment. There has been a recent proliferation of data-flow languages such as Sawzall [73], DryadLINQ [103], HiveQL [94], and Pig Latin [70] for processing structured data on parallel data processing systems such as Hadoop. Another such query language, JAQL [19] is designed for semi-structure data analytics, and uses the (key, value) JSON model. However, this model splits RDF sub graphs into different bags, and may not be efficient to execute bushy plans. A previous work, RAPID [89] focused on optimizing analytical processing of RDF data on Pig. The work in [76] extends Pig with User Defined Functions (UDF) to enable TripleGroup-based processing. This chapter provides formal semantics to integrate TripleGroups as first-class citizens, and presents operators for graph pattern matching.

RDF Data processing on MapReduce Platforms. MapReduce framework has been explored for scalable processing of Semantic Web data. For reasoning tasks, specialized map and reduce functions have been defined based on RDFS rules [96] and the OWL Horst rules [95], for materializing the closure of RDF graphs. Yet another work [93] extends Pig by integrating schema-aware RDF data loader and embedding reasoning support into the existing framework. For scalable pattern matching queries, there have been MapReduce-based storage and query systems [65, 66] that process RDFMolecules. Another work [47] pre-processes RDF triples to enable efficient querying of billions of triples over HDFS. SHARD [82] uses initial MR cycles to cluster triples into star subgraphs, followed by separate MR cycles to process each clause in the SPARQL query. Another system [9] uses HadoopDB [8] with a column-oriented database to support a scalable Semantic Web application. This framework enables parallel computation of star-joins if the data is partitioned based on the Subject component. The HadoopDB-based extension [45] uses a hybrid database-Hadoop architecture that exploits the partitioning scheme to push part of the execution into the database/RDF-3X. However, once the execution is handed over to Hadoop each join would be evaluated in a separate MR cycle. HadoopRDF [46] pre-processes triples using the vertical-partitioning (VP) [6] approach, and uses heuristics to greedily group non-conflicting joins in a query to minimize the required number of MR cycles. We focus on ad hoc processing of RDF graphs that cannot presume pre-processed or indexed data.

Optimizing Multi-way Joins. RDF graph pattern matching typically involves several join operations. There have been optimization techniques [98] to re-write SPARQL queries into small-sized star-shaped groups and generate bushy plans using two physical join operators called njoin and gjoin. It is similar in spirit to the work presented here since both exploit star-shaped

sub patterns. However, our work focuses on parallel platforms and uses a grouping-based algorithm to evaluate star-joins. There has been work on optimizing m-way joins on structured relations like slice join [56]. However, we focus on joins involving RDF triples for semi-structured data. Another approach [10] efficiently partitions and replicates tuples across reducers in a way that minimizes the communication cost. Another multi-way join algorithm [102] uses a replicated scheme to clusters joins into few MR cycles. This is complementary to our approach and the partitioning schemes could further improve the performance of join operations. [23] investigates several join algorithms which leverage pre-processing techniques on Hadoop, but mainly focus on log processing. RDFBroker [88] is a RDF store that is based on the concept of a *signature* (set of properties of a resource), similar to NTGA's *structure-labeling* function λ. However, the focus of [88] is to provide a natural way to map RDF data to database tables, without presuming schema knowledge. Pregel [61] and Signal/Collect [92] provide graph-oriented primitives as opposed to relational algebra type operators, and also target parallel platforms.

2.1.2 Contributions

This work proposes an approach for increasing the degree of parallelism by enabling some form of *inter-operator parallelism*. This allows us to "sneak in" bushy like query execution plans into Hadoop by interpreting star-joins as groups of triples or *TripleGroups*. We provide the foundations for supporting TripleGroups as first class citizens. We introduce an intermediate algebra called the *Nested TripleGroup Algebra* (NTGA) that consists of TripleGroup operators as alternatives to relational style operators. We also present a data representation format called the *RDFMap* that allows for a more easy-to-use and concise representation of intermediate query results than the existing format targeted at relational tuples. RDFMap aids in efficient management of schema-data associations, which is important while querying *schema-last* data models like RDF. Specifically, we propose the following:

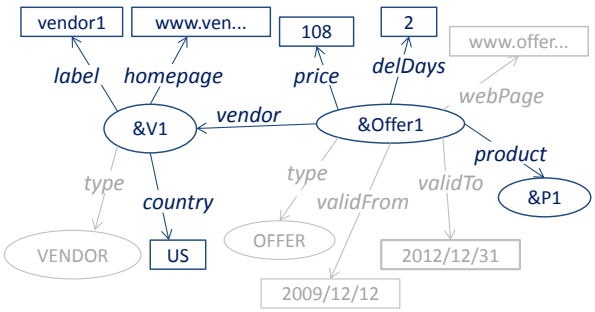
- A TripleGroup data model and an algebra called the *Nested TripleGroup Data Model and Algebra* (NTGA) [77], that leads to efficient MapReduce execution plans with reduced numbers of MapReduce cycles.

- Operator implementation strategies that are integrated into our system *RAPID+* [51] (NTGA-based extension of Apache Pig) to minimize the costs involved in RDF graph processing on MapReduce platforms.

- Establishing an equivalence between expressions in NTGA and relational algebra expressions representing a class of graph pattern queries. This equivalence is established by introducing a notion of *content equivalence*.

Subject	Property	Object
&Offer1	*type*	OFFER
&Offer1	*price*	108
&Offer1	*delDays*	2
&Offer1	*vendor*	&V1
&Offer1	*product*	&P1
&Offer1	*validTo*	2012/12/31
&V1	*type*	VENDOR
&V1	*country*	US
&V1	*label*	"vendor1"
&V1	*homepage*	www.ven...

(a)



(b)

Figure 2.1: Example RDF data with details about product Offers and their vendors represeted as (a) a triple relation with (Subject, Property, Object) (b) a graph with (node, edge, node)

- A comparative performance evaluation of RAPID+ using a benchmark dataset shows up to 60% performance improvement over the default Pig system for certain classes of graph pattern queries.

This chapter is organized as follows: Section 2.2 reviews the basics of RDF graph pattern matching, and the issues involved in processing such pattern queries in MapReduce based systems like Pig. This section also summarizes the optimization strategies presented in a previous work [76] which form a base for the algebra proposed in this paper. Section 2.3 presents the TripleGroup data model and the supported operations, and the integration of NTGA operators into Apache Pig. Section 2.4 presents the evaluation results comparing the performance of RAPID+ with the default Pig system.

2.2 Relational-style Processing of Graph Pattern Queries on MapReduce

An RDF database is a collection of statements describing entities or *triples* of the form ($Subject$, $Property$, $Object$) where the $Property$ defines a binary relationship between entities (denoted by leading '&') or between entities and their attributes. Figure 2.1(a) shows example RDF triples from the Berlin SPARQL Benchmark [22] dataset which describes product Offers and their Vendors. For example, the triple ($\&Offer1, vendor, \&V1$) states that the resource $\&Offer1$ relates to a vendor entity $\&V1$. A collection of RDF triples can also be modeled as a directed labeled graph as shown in Figure 2.1(b) with nodes representing Subjects and Objects, and labeled edges denoting the Property types. Both nodes and edges in an RDF graph are Semantic Web resources identified by URIs, but objects may be literals.

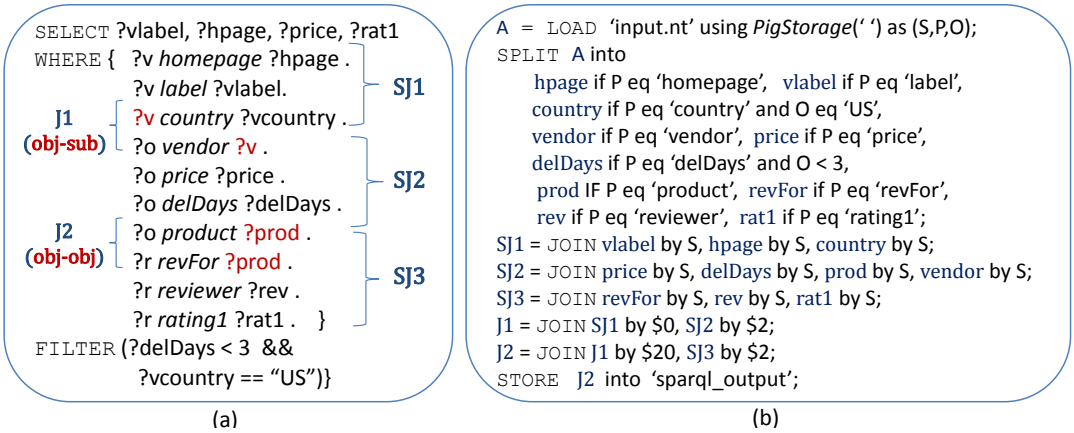


Figure 2.2: Example pattern matching query in (a) SPARQL (b) Pig Latin (VP approach)

2.2.1 RDF Graph Pattern Matching

The foundational construct for querying an RDF graph is a *triple pattern*, which is a triple with a variable (denoted by leading '?') in any of the Subject, Property, or Object positions. For example, the triple pattern $(?s1, product, ?o1)$ matches all triples with property *product*, whose Subjects and Objects are considered as variable bindings for $?s1$ and $?o1$, respectively. Typical queries on an RDF database involve a *Basic Graph Pattern*, which is a conjunction of two or more triple patterns where shared variables denote equi-joins. A graph pattern is equivalent to the select-project-join (SPJ) construct in SQL. The answer to a graph pattern consists of bindings that match all triple patterns. Consider an example query on the BSBM[1] data graph to retrieve "*the details of US-based vendors who deliver products within three days, along with the review details for these products*". Figure 2.2 (a) shows the corresponding SPARQL query with 10 triple patterns. Such a query can be partitioned based on the common subject variables $?v$, $?o$, and $?r$ into three star-join structures $SJ1$, $SJ2$, $SJ3$, that describe resources of type Vendor, Offer, and Review, respectively. The query also consists of two chain-join patterns ($J1$, $J2$) to combine these star patterns and a filter construct to restrict answers to those products from US-based vendors with delivery days less than 3.

There are two main ways of processing RDF graph patterns depending on the storage model used: (i) triple model, or (ii) vertically partitioned (VP) [6] storage model in which the triple relation is partitioned based on properties. In the former approach, RDF pattern matching queries can be processed as series of relational style self-joins on a large triple relation. Some systems use multi-indexing schemes [64] to counter this bottleneck. The VP approach results in a series of join operations but on smaller property-based relations. Another observation [64]

[1] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/

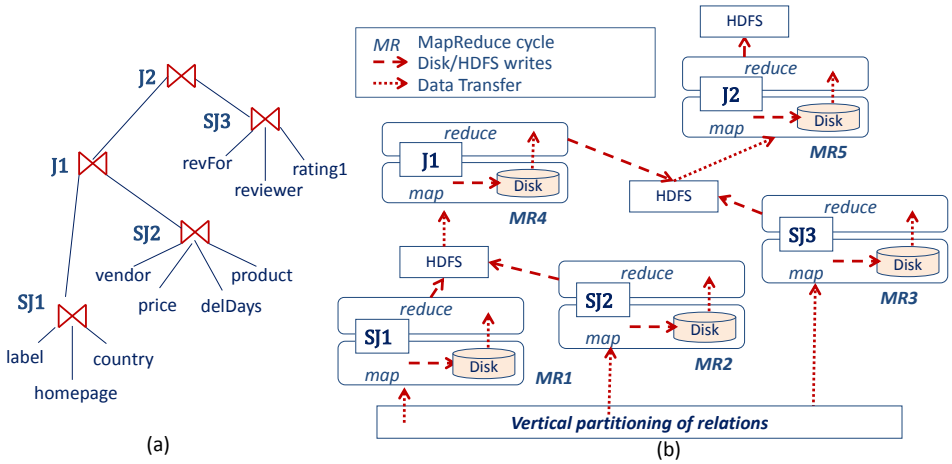


Figure 2.3: Pattern Matching using VP approach (a) Relational algebra query plan (b) Map-Reduce execution flow

is that graph pattern matching queries on RDF data often consist of multiple star-structured graph sub patterns. For example, 50% of the benchmark queries in BSBM have at least two or more star patterns. Existing work [64] [98] optimize pattern matching by exploiting these star-structures to generate bushy query plans.

2.2.2 Graph Pattern Matching in Apache Pig

MapReduce data processing platforms like Pig focus on ad hoc data processing in the cloud environment where the existence of preprocessed and suitably organized data cannot be presumed. Therefore, in the context of RDF graph pattern processing which is done directly from input documents, the VP approach with smaller relations is more suitable. To capture the VP storage model in Pig, an input triple relation needs to be "split" into property-based partitions using Pig Latin's `SPLIT` command. Then, the star-structured joins are achieved using an m-way `JOIN` operator, and chain joins are executed using the traditional binary `JOIN` operator. Figure 2.2(b) shows how the graph pattern query in Figure 2.2(a) can be expressed and processed in Pig Latin. Figure 2.3 (a) shows the corresponding query plan for the VP approach. We refer to this sort of query plan as Pig's approach in the rest of the chapter. Alternative plans may change the order of star-joins based on cost-based optimizations. However, that issue does not affect our discussion because the approaches compared in this chapter benefit similarly from such optimizations. Pig Latin queries are compiled into a sequence of Map-Reduce (MR) jobs that run over Hadoop. The Hadoop scheduling supports *partition parallelism* such that in every stage, one operator is running on different partitions of data at different nodes. This leads to a linear style physical execution plan. The above logical query plan will be compiled into a linear

execution plan with a sequence of five MR cycles as shown in Figure 2.3 (b). Each join step is executed as a separate MR job. However, Pig optimizes the multi-way join on the same column, and compiles it into a single MR cycle.

Issues. (i) Each MR cycle involves communication and I/O costs due to the data transfer between the Mappers and Reducers. Intermediate results are written to disk by Mappers after the *map* phase, which are read by Reducers and processed in the *reduce* phase after which the results are written to HDFS (Hadoop Distributed File System). These costs are summarized in Figure 2.3 (b). Using this style of execution where join operations are executed in different MR cycles, join-intensive tasks like graph pattern matching will result in significant I/O and communication overhead. There are other issues that contribute I/O costs e.g. the `SPLIT` operator for creating VP relations generates concurrent sub flows which compete for memory resources and is prone to disk spills. (ii) In imperative languages like Pig Latin, users need to explicitly manipulate the intermediate results. In *schema-last* data models like RDF, there is an increased burden due to the fact that users have to keep track of which columns of data are associated with which schema items (*properties*) as well as their corresponding values. For example, for the computation of join $J2$, the user needs to specify the join between intermediate relation $J1$ on the value of property type "product", and relation $SJ3$ on the value of property type "reviewFor". It is not straightforward for the user to determine that the value corresponding to property "product" is in column 20 of relation $J1$. In *schema-first* data models, users simply reference desired columns by attribute names.

Intuition of Approach. All star subgraphs matching the star subpatterns in a query can be retrieved in a single MR cycle by re-interpreting star-joins using a *grouping-based join algorithm*. It can be observed that performing a group by on Subject column of the triple relation, yields 'groups of tuples' or *TripleGroups* that represent all the star sub graphs in the database. We can obtain all these star sub graphs using the relational style `GROUP BY` which executes in a *single* MR cycle, thus minimizing the overall I/O and communication overhead in RDF graph processing. Additionally, repeated data processing costs can be improved by *coalescing* operators in a manner analogous to "pushing select into cartesian product" in relational algebra to produce a more efficient operator.

The next section presents a generalization of this strategy by proposing an intermediate algebra based on the notion of TripleGroups. This provides a formal foundation to develop first-class operators with more precise semantics, to enable tighter integration into existing systems to support automatic optimization opportunities. A more suitable data representation format is also proposed to aid in efficient and user-friendly management of intermediate results of operators in this algebra. This representation scheme is used to implement the proposed operators.

2.3 Foundations

2.3.1 Data Model and Algebra

Nested TripleGroup Algebra (NTGA) is based on the notion of the *TripleGroup* data model which is formalized as follows:

Definition 1. (*TripleGroup*) A TripleGroup tg is a relation of triples t_1,t_2,...t_k, whose schema is defined as (S, P, O). Further, any two triples t_i, $t_j \in tg$ have overlapping components i.e. t_i [col_i] = t_j[col_j] where col_i, col_j refer to subject or object component. When all triples agree on their subject (object) values, we call them *subject* (*object*) *TripleGroups* respectively. Figure 2.4 (a) is an example of a *subject TripleGroup* which corresponds to a star sub graph. Our data model allows triplegroups to be nested at the object component.

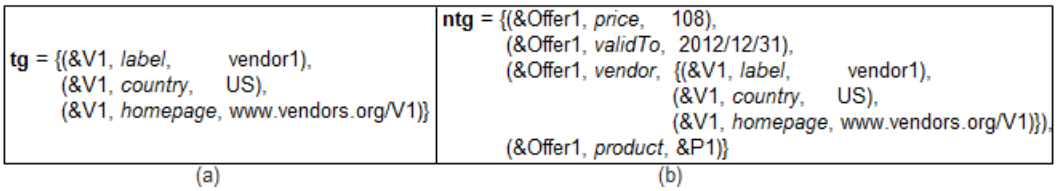


Figure 2.4: (a) Subject TripleGroup tg (b) Nested TripleGroup ntg

Definition 2. (*Nested TripleGroup*) A nested TripleGroup ntg consists of a root triplegroup ntg.`root` and one or more child triplegroups returned by the function ntg.`child`() such that: For each child triplegroup $ctg \in ntg$.`child`(),

- $\exists\ t_1 \in ntg$.`root`, $t_2 \in ctg$ such that t_1.`Object` = t_2.

Nested triplegroups capture the graph structure in an RDF model in a more natural manner. An example of a nested triplegroup is shown in Figure 2.4 (b). A nested triplegroup can be "unnested" into a flat triplegroup using the `unnest` operator. The definition is shown in Figure 2.6. Figure 2.5 (a) shows the triplegroup resulting from the `unnest` operation on the nested triplegroup in Figure 2.4 (b). In addition, we define the `flatten` operation to generate an "equivalent" n-tuple for a given triplegroup. For example, if tg = t_1, t_2,..., then the n-tuple tu has triple t_1= ($s1$, $p1$, $o1$) stored in the first three columns of tu, triple t_2 = ($s2$, $p2$, $o2$) is stored in the fourth through sixth column, and so on. For convenience, we define the function `triples`() to extract the triples in a triplegroup. For the triplegroup in Figure 2.4 (a), the `flatten` is computed as tg.`triples`($label$) $\bowtie$ tg.`triples`($country$) $\bowtie$ tg.`triples`($homepage$), resulting in an n-tuple as shown in Figure 2.5 (b). It is easy to observe that the information content in both formats is equivalent. We refer to this kind of equivalence as *content equivalence* which we will denote as $\cong$. Consequently, computing query results in terms of triplegroups is lossless in terms of information. This is specifically important in scenarios where triplegroup-based processing is

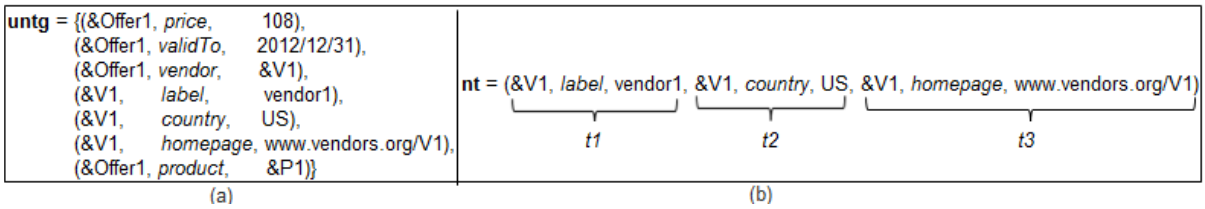


Figure 2.5: (a) ntg.`unnest`() (b) n-tuple after tg.`flatten`()

more efficient.

We define other triplegroup functions as shown in Figure 2.6 (b). The *structure-labeling function* λ assigns each triplegroup tg, with a label that is constructed as some function of tg.`props`(). Further, for two triplegroups tg_1, tg_2 such that tg_1.`props`() $\subseteq$ tg_2.`props`(), λ assigns labels such that $tg_1.\lambda() \subseteq tg_2.\lambda()$. The labeling function λ induces a partition on a set of triplegroups based on the structure represented by the property types present in that triplegroup. Each equivalence class in the partition consists of triplegroups that have the exact same set of property types.

Next, we discuss some of the triplegroup operators which are formally defined in Figure 2.6(c).

`TG_Proj` (π^{γ}). The triplegroup project operator extracts from each triplegroup, the required triple component from the triple matching the triple pattern. In the case of our example, we have $\texttt{TG\_Proj}_{?hpage}(\text{TG}) = \{\ www.vendors.org/V1\ \}$.

`TG_Filter` (σ'). The triplegroup filter operator is used for *value-based filtering* i.e., to check if the triplegroups satisfy a given filter condition. For example, $\texttt{TG\_Filter}_{price>500}(TG)$ would eliminate the triplegroup ntg in Figure 2.4 (b) since the triple ($\&Offer1$, $price$, 108) does not satisfy the filter condition.

`TG_GroupFilter` (σ^{γ}). The group-filter operation is used for *structure-based filtering* i.e., to retain only those triplegroups that satisfy the structural constraints specified by the query sub structure. For example, the `TG_GroupFilter` operator can be used to eliminate triplegroups like tg in Figure 2.4 (a), that are structurally incomplete with respect to the equivalence class $TG_{\{label,country,homepage,mbox\}}$.

`TG_Join` ($\bowtie^{\gamma}$). The join expression $\texttt{TG\_Join}(?v_{tp_x}{:}TG_x, ?v_{tp_y}{:}TG_y)$ computes the join between a triplegroup tg_x in equivalence class TG_x with a triplegroup tg_y in equivalence class TG_y based on the specified triple patterns. The triple patterns tp_x and tp_y share a common variable $?v$ at O or S component. The result of an object-subject (O-S) join is a nested triplegroup in which tg_y is nested at the O component of the join triple in tg_x. For example, Figure 2.7 (bottom) shows the nested triplegroup resulting from the join operation between equivalence classes $TG_{\{price,validTo,vendor\}}$ and $TG_{\{label,country,homepage\}}$ that join based on triple patterns $\{?o\ vendor\ ?v\}$ and $\{?v\ country\ ?o2\}$ respectively. For object-object (O-O) joins, the triplegroup join operator computes a triplegroup by union of triples in the individual triplegroups.

Symbol	Description
tg	TripleGroup
TG	Set of TripleGroups
tp	Triple pattern
ntg.root	Root of the nested triplegroup
ntg.child()	Children of the nested triplegroup
$?v_{tp}$	A variable in the triple pattern tp

(a)

Function	Returns
tg.props()	Union of all property types in tg
tg.triples()	Union of all triples in tg
tg.triples(p_i)	Triples in tg with property type p_i
$tg.\lambda()$	Structure label for tg based on tg.props()
$\delta(tp)$	A triple matching the triple pattern tp
$\delta(?v_{tp})$	A variable substituion in the triple matching tp

(b)

Operator	Definition
TG_Load($\{t_i\}$)	$\{ tg_i \mid tg_i = t_i$, and t_i is an input triple$\}$
TG_Proj$_{?v_{tp}}(TG)$	$\{\delta_i(?v_{tp}) \mid \delta_i(tp) \in tg_i$, $tg_i \in TG$ and $tp.\lambda() \subseteq tg_i.\lambda()\}$
TG_Filter$_{\Theta(?v_{tp})}(TG)$	$\{ tg_i \mid tg_i \in TG$ and $\exists\ \delta_i(tp) \in tg_i$ such that $\delta_i(?v_{tp})$ satisfies the filter condition $\Theta(?v_{tp})\}$
TG_GroupFilter(TG, P)	$\{ tg_i \mid tg_i \in TG$ and tg_i.props() $= P \}$
TG_Join($?v_{tp_x}$:TG_x, $?v_{tp_y}$:TG_y)	Assume $tg_x \in TG_x$, $tg_y \in TG_y$, $\exists\ \delta_1(tp_x) \in tg_x$, $\delta_2(tp_y) \in tg_y$, and $\delta_1(?v_{tp_x}) = \delta_2(?v_{tp_y})$ if O-S join, then $\{ntg_i \mid ntg_i$.root $= tg_x$, $\delta_1(tp_x)$.Object $= tg_y\}$ else $\{ tg_x \cup tg_y\}$
tg.flatten()	$\{tg$.triples(p_1) $\bowtie tg$.triples(p_2)...$\bowtie tg$.triples(p_n) where $p_i \in tg$.props()$\}$
ntg.unnest()	$\{ t_i \mid t_i$ is a non-nested triple in tg.root $\}$ $\cup \{ (s, p, s') \mid t' = (s, p, (s', p', o'))$ is a nested triple in tg.root$\}$ $\cup \{ ctg_i$.unnest() $\mid ctg_i \in tg$.child() $\}$

(c)

Figure 2.6: NTGA Quick Reference (a) Symbols (b) Functions (c) Operators

2.3.2 MapReduce Execution plan using NTGA and its mapping to Relational Algebra.

Consider a query Q over a triple relation T with two star subpatterns Stp_1 and Stp_2 such that the set of properties, Stp_1.props() $= P_{stp_1} = \{price, validTo, vendor\}$ and Stp_2.props() $= P_{stp_2} = \{label, country, homepage\}$. The corresponding NTGA based query plan is represented in Figure 2.7, which we describe below.

- TG_LoadFilter loads triples in T and retains only a subset T_Q that is relevant to the query i.e. triples whose properties do not match either P_{stp_1} or P_{stp_2} are eliminated (property-based filtering).

- TG_GroupBy (γ_{Sub}) groups the triples in T_Q based on the subject column to produce a set of subject tripleGroups TG. Due to property-based filtering, a triplegroup $tg \in TG$ only contains triples that are relevant to either stp_1 or stp_2.

- TG_GroupFilter (σ^{γ}) ensures strict graph pattern matching based on the bound-properties in the star-patterns and eliminates triplegroups that contain missing edges (structure-based filtering). Given a set of triplegroups TG and a set of bound-properties P_{stp_1},

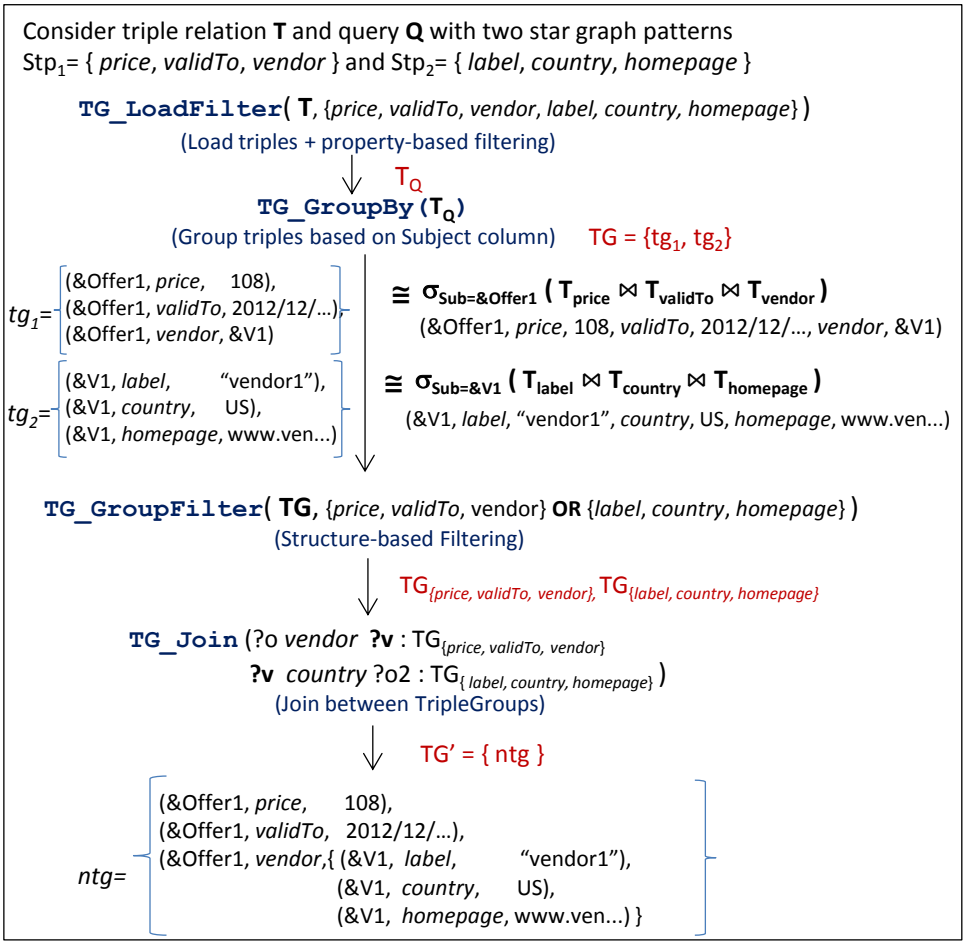


Figure 2.7: NTGA-based processing of graph pattern queries

`TG_GroupFilter` retains the subset of triplegroups TG_{stp_1} that contain triples matching each of the properties in P_{stp_1}. Thus, `TG_GroupFilter` produces triplegroups that are valid answers to either stp_1 or stp_2. Any other triplegroup is eliminated in this phase.

- Triplegroups thus produced are '*content-equivalent*' (represented as $\cong$) to the set of n-tuples computed using a set of relational m-way star-joins. For example,

$$tg_1 \cong (\ \&Offer1,\ price,\ 108,\ validTo,\ 2012/12/,\ vendor,\ \&V1\ ) \in Tup_{stp_1}$$

Further, triplegroup $tg_1 \in TG_{\{price,validTo,vendor\}}$ can be flattened into its equivalent set of n-tuples using:

$$tg_1.\texttt{triples}(price) \bowtie tg_1.\texttt{triples}(validTo) \bowtie tg_1.\texttt{triples}(vendor)$$
$$= (\ \&Offer1,\ price,\ 108,\ validTo,\ 2012/12/,\ vendor,\ \&V1\ )$$

In general, for $P_{stp_1} = \{P_1, P_2, ..., P_k\}$ and $tg \in TG_{Stp_1}$, the `flatten` operator partitions the triples in tg based on its properties, and joins the k-partitions to produce k-tuples $\{(t_1, t_2, ..., t_k)\} \in Tup_{stp_1}$.

- Next, `TG_Join` ($\bowtie^{\gamma}$) is used for joins between triplegroups. For example, the object-subject join between $tg_1 \in TG_{stp_1}$ and $tg_2 \in TG_{stp_2}$ results in a nested triplegroup ntg whose root is the triplegroup tg_1 and child triplegroup is tg_2.

- Nested triplegroups resulting from `TG_Join` between the set of triplegroups TG_{stp_1} and TG_{stp_2} are content-equivalent to the corresponding n-tuples computed using a relational join between the intermediate star-joined relations Tup_{stp_1} and Tup_{stp_2} respectively.The nested triplegroup ntg can be flattened by flattening the root as well as the child triplegroup as: $\{tg_1.\texttt{flatten}() \bowtie tg_2.\texttt{flatten}()\}$.

In general, the equivalence mapping between the relational algebra and the NTGA can be summarized as follows:

Star-join Computation. Consider a triple relation T and let $Stp_1 = \{ P_1, P_2,...,P_k\}$ be the required star pattern in the query. Let T_{P_i} represent a vertically partitioned (VP) relation with triples involving property P_i. Then, the relational-style star-join computation involves joining the relevant VP relations. In NTGA, the required star-join is computed by first grouping the triples based on the subject column and then enforcing the required structural constraints using the triplegroup group-filter operator, i.e.,

$$(T_{P_1} \bowtie T_{P_1} \bowtie ... \bowtie T_{P_k}) \cong \sigma^{\gamma}_{\{P_1,P_2,..,P_k\}}(\gamma_{Sub} (T)) = TG_{\{P_1,P_2,...,P_k\}}$$

where $TG_{\{P_1,P_2,...,P_k\}}$ represents the set of triplegroups that match the given star pattern.

Join between Stars. Consider a graph pattern involving a join between two star patterns $Stp_1 = \{ P_1, P_2,...,P_k\}$ and $Stp_2 = \{ P_{1'}, P_{2'},...,P_{m'}\}$. Then, the join between the stars can be expressed in NTGA using the triplegroup join operator, i.e.,

$$(T_{P_1} \bowtie ... \bowtie T_{P_k}) \bowtie (T_{P_{1'}} \bowtie ... \bowtie T_{P_{m'}}) \cong \bowtie^{\gamma}(TG_{\{P_1,P_2,...,P_k\}}, TG_{\{P_{1'},P_{2'},...,P_{m'}\}})$$

The NTGA based graph pattern matching for a query with n star sub patterns, compiles into a MapReduce flow with n MR cycles as shown in Figure 2.8. The same query executes in double the number of MR cycles $(2n-1)$ using relational-style query plans in systems such as Apache Pig. Figure 2.8 shows the equivalence between the NTGA and relational algebra operators based on our notion of *content equivalence*. This mapping suggests rules for lossless transformation between queries written in relational algebra and NTGA.

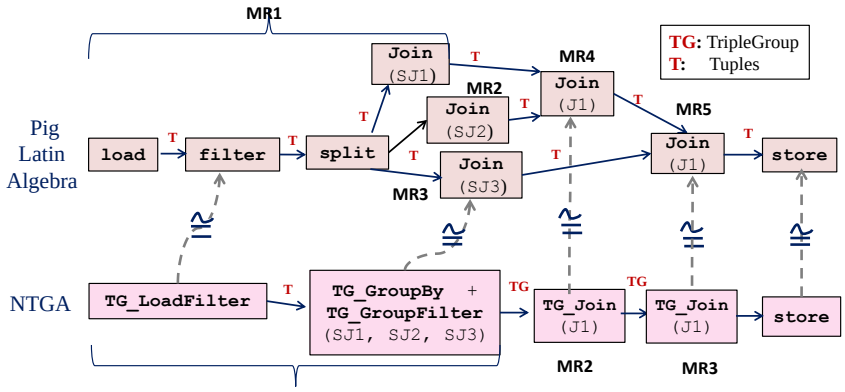


Figure 2.8: NTGA execution plan and mapping to Relation Algebra

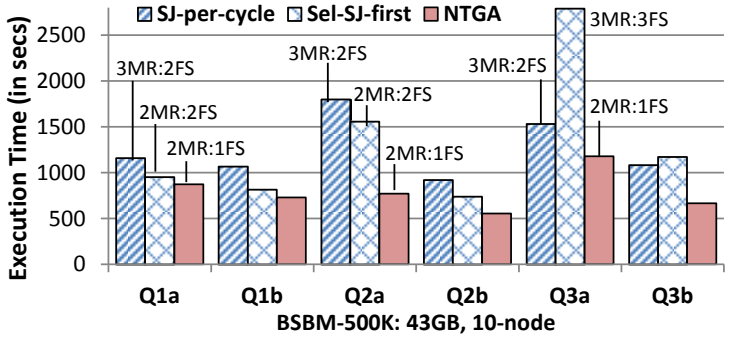


Figure 2.9: Evaluation of different groupings of star-joins (MR: No. of MapReduce cycles, FS: No. of Full Scans)

2.3.3 Benefit of NTGA Query Plans.

As seen in the previous section, for a query with 'n' star subpatterns, NTGA can compute ALL star subpatterns concurrently using a single 'grouping' operation, by first 'grouping' the triples into subject triplegroups and then applying a disjunctive selection based on the multiple star subpatterns. This is in contrast to the relational-style approach where each star subpattern is evaluated as a relational-style join. The grouping-based star-join computation naturally fits the *map-group-reduce* theme in MapReduce, and translates to just one MR cycle for computing all star-joins in the query (as opposed to 'n' MR cycles using relational-style plans).

Consider a case study using 6 test queries (each with two star subpatterns) using the BSBM synthetic benchmark dataset (43GB) on a 10-node Hadoop cluster, as shown in Figure 2.9. The test queries have varying join structures with Object-Subject join (Q1a, Q1b, Q2a, Q2b) and Object-Object join (Q3a, Q3b) between star patterns. Queries Q1b, Q2b, Q3b are variations of Q1a, Q2a, Q3a respectively, where one of the two star-joins is highly selective due to an additional filter on the object column. The evaluated queries and the Hive scripts are available in Appendix A.1.

We evaluated three different groupings of star subpatterns in a query, (i) a star-join per cycle approach (*SJ-per-cycle*), (ii) most selective grouping of joins first but preserving star structure as much as possible to minimize MR cycles (*Sel-SJ-first*), and (iii) concurrent evaluation of star-joins using the grouping-based approach in NTGA. *SJ-per-cycle* approach requires 3 MR cycles for all queries (2 of 3 cycles require full scan of triple relation). For Object-Subject joins, *Sel-SJ-first* approach can group joins into just 2 MR cycles (both cycles scan entire triple relation). For the Object-Object join (Q3a, Q3b), *Sel-SJ-first* still requires 3 MR cycles, but more importantly has very high HDFS reads due to full scan of triple relation in all 3 cycles. In contrast, the *NTGA* approach is able to minimize the number of MR cycles (2 cycles for all queries), as well as minimize the required number of full scans of the triple relation, thus outperforming the other two approaches for all test queries.

2.3.4 Integrating NTGA operators into Apache Pig

Data Structure for TripleGroups - RDFMap.

Pig Latin data model supports a bag data structure that can be used to capture a triplegroup. The Pig data bag is implemented as an array list of tuples and provides an iterator to process them. Consequently, implementing NTGA operators such as `filter`, `groupfilter`, `join` etc. using this data structure requires an iteration through the data bag which is expensive. For example, given a graph pattern with a set of triple patterns TP and a data graph represented as a set of triplegroups TG, the `groupfilter` operator requires matching each triple pattern in TP with each tuple t in each triplegroup $tg \in TG$. This results in the cost of the `groupfilter` operation being $O(|TP|*|tg|*|TG|)$. In addition, representing triples as 3-tuple (s, p, o) results in redundant $s(o)$ components for subject (object) triplegroups. We propose a specialized data structure called *RDFMap* targeted at efficient implementation of NTGA operators. Specifically it enables, (i) efficient look-up of triples matching a given triple pattern, (ii) compact representation of intermediate results, and (iii) ability to represent structure-label information for triplegroups. RDFMap is an extended HashMap that stores a mapping from property to object values. Since subject of triples in a triplegroup are often repeated, RDFMap avoids this redundancy by using a single field Sub to represent the subject component. The field EC captures the structure-label (equivalence class mapped to numbers). Figure 2.10. shows the RDFMap corresponding to the Subject triplegroup in Figure 2.4 (a). Using this representation model, a nested triplegroup can be supported using a nested $propMap$ which contains another RDFMap as a value. The $propMap$ provides a property-based indexed structure that eliminates the need to iterate through the tuples in each bag. Since $propMap$ is hashed on the P component of the triples, matching a triple pattern inside a triplegroup can now be computed in time $O(1)$. Hence, the cost of the `groupfilter` operation is reduced to $O(|P|*|TG|)$.

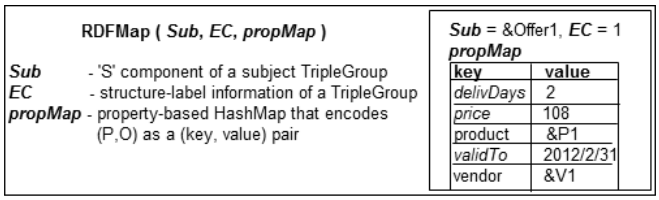


Figure 2.10: RDFMap representing a subject triplegroup

Implementing NTGA operators using RDFMap.

In this section, we show how the property-based indexing scheme of an RDFMap can be exploited for efficient implementation of the NTGA operations. We then discuss the integration of NTGA operators into Pig.

StarGroupFilter. A common theme in our implementation is to *coalesce* operators where possible in order to minimize the costs of parameter passing, and context switching between methods. The `StarGroupFilter` is one such operator, which coalesces the NTGA `groupfilter` operator into Pig's relational `GROUP BY` operator. Creating subject triplegroups using this operator can be expressed as:

$$TG = \texttt{StarGroupFilter}\ \text{triples by}\ S$$

Algorithm 1: StarGroupFilter

```
   Map (key:null, val: Tuple tup(s,p,o)) ;
1  emit ⟨s, tup⟩ ;
   Reduce (key:Subject Sub, val:List of tuples T) ;
   //locBitSet - record property types in T
   //ECList - list of global BitSets for all ECs
2  foreach tup(s, p, o) ∈ T do
3      set p in locBitset ;
4      add (p,o) to tempMap ;
5  matchedECList ← match(locBitSet, ECList);
6  foreach EC ∈ matchedECList do
7      propMap ← extract subset of tempMap based
                 on properties in EC;
8      emit ⟨RDFMap(Sub, EC, propMap)⟩;
```

The corresponding *map* and *reduce* functions for the `StarGroupFilter` operator (NTGA's `TG_GroupBy` followed by TG_GroupFilter) are shown in Algorithm 1. In the *map* phase, the tuples are annotated based on the subject component s (line 1), analogous to the *map* of a `GROUP BY` operator. In the *reduce* function, the different tuples sharing the same subject component are packaged into an *RDFMap* that corresponds to a subject triplegroup. The `groupfilter` operator is integrated into the reduce of `StarGroupFilter` for *structure-based filtering* based on the query sub structures (equivalence classes). This is achieved using global bit patterns (stored as BitSet) that concisely represent the property types in each equivalence class. As the tuples

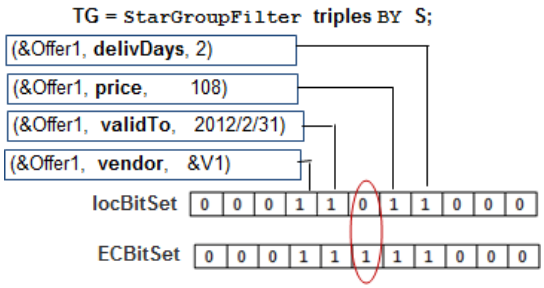


Figure 2.11: Structure-based filtering of triplegroups

are processed in the *reduce* function, the local BitSet (`locBitSet` in line 3) keeps track of the property types processed. The processed (property, object) pairs are stored in a temporary map (*tempMap* in line 4). After processing all tuples in a group, the local bit set is matched with the global BitSets corresponding to the required structures relevant to the query in question (line 5). Note that the group of tuples may contain subsets of tuples that match more than one relevant structure (denoted as *matchedECList* in line 5). For each matching equivalence class *EC*, a new *RDFMap* containing the relevant (*p*,*o*) pairs from *tempMap* is produced (lines 6-8). If the local BitSet (`locBitSet`) does not match the global BitSet (`ECBitSet`), the structure is incomplete and the group of tuples is eliminated. Figure 2.11 shows the mismatch between the `locBitSet` and `ECBitSet` in the sixth position that represents the missing property "product" belonging to the equivalence class $TG_{\{price,validTo,delivDays,vendor,product\}}$. The output of `StarGroupFilter` is a single relation containing a list of RDFMaps corresponding to the different star sub graphs in the input data.

RDFJoin. The `RDFJoin` operator takes as input a single relation containing RDFMaps, and computes the NTGA `join` between star patterns. The object-subject join $J1$ between the star patterns in Figure 2.2 can be expressed as follows:

$$J1 = \texttt{RDFJoin}\ TG \text{ on } (1{:}\text{`}vendor\text{'},\ 0{:}*);$$

where joins on object are specified using the property and joins on subject are specified as '*'. Algorithm 2 shows the map and reduce functions for the `RDFJoin` operator (NTGA's join operator). In the *map* phase, the RDFMaps are annotated based on the join key corresponding to their equivalence class (lines 1-5). In the *reduce* phase, the RDFMaps are separated based on their equivalence class *EC* (lines 5-8). The RDFMaps that join are packaged into a new RDFMap (lines 11-15) which corresponds to a nested triplegroup. For example, the `RDFJoin` between triplegroups shown in Figure 2.7, results in an RDFMap whose *propMap* contains the union of triples from the individual triplegroups as shown in Figure 2.12. The equivalence class *EC* of the new joined RDFMap is a function of the *EC* of the individual RDFMaps (line 12). In our implementation, the *Sub* field is a concatenation of the *Sub* fields of the joining

Algorithm 2: RDFJoin

```
   Map (key:null, val: RDFMap rMap) ;
1  if join on Sub then
2      joinKey ← rMap.Sub;
   else if join on Obj then
3      joinKey ← extract joinKey from rMap.propMap;
4  emit ⟨ joinKey, rMap⟩ ;
   Reduce (key:joinKey, val:List of RDFMaps R) ;
5  foreach rMap ∈ R do
6      if rMap.EC == EC1 then
7          add rMap to leftList;
       else if rMap.EC == EC2 then
8          add rMap to rightList;
9  foreach left ∈ leftList do
10     foreach right ∈ rightList do
11         propMapNew ← joinProp(left.propMap, right.propMap);
12         ECNew ← joinEC(left.EC, right.EC);
13         SubNew ← joinSub(left.Sub, right.Sub);
14         rMapNew ← RDFMap(SubNew, ECNew, propMapNew);
15         emit ⟨rMapNew⟩;
```

RDFMaps such as $\&Offer1.\&V1$ in our example. Our join result corresponds to an unnested joined triplegroup, as shown in Figure 2.5(a).

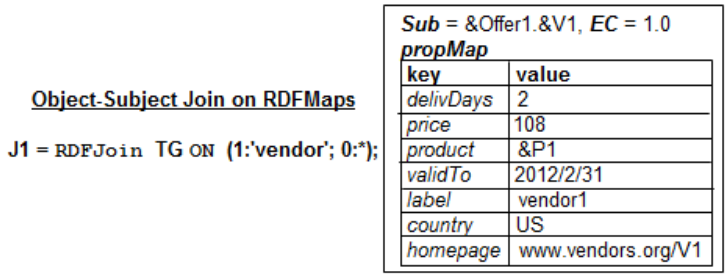


Figure 2.12: Example RDFMap after RDFJoin operation

2.4 Empirical Evaluation

Our goal was to empirically evaluate the performance of NTGA operators with respect to pattern matching queries involving combinations of star and chain joins. We compared the performance of RAPID+ with two implementations of Pig, (i) the naive Pig with the VP storage model, and (ii) an optimized implementation of Pig (Pig_{opt}), in which we introduced additional project operations to eliminate the redundant join columns. Our evaluation tasks included, (i) **Task1 -** Scalability of TripleGroup-based approach with size of RDF graphs, (ii)

Table 2.1: Testbed queries and performance comparison of RAPID+ with Pig (32GB, 10-node)

Query	No. of Triple Patterns	No. of Edges in Stars	%gain	Query	No. of Triple Patterns	No. of Edges in Stars	%gain
Q1	3	1:2	56.8	Q6	8	4:4	58.4
Q2	4	2:2	46.7	Q7	9	5:4	58.6
Q3	5	2:3	47.8	Q8	10	6:4	57.3
Q4	6	3:3	51.6	2S1C	6	2:4	65.4
Q5	7	3:4	57.4	3S2C	10	2:4:4	61.5

Task2 - Scalability of TripleGroup-based pattern matching with denser star patterns, and (iii) **Task3 -** Scalability of NTGA operators with increasing cluster sizes.

2.4.1 Setup

Environment: The experiments were conducted on VCL [83], an on-demand computing and service-oriented technology that provides remote access to virtualized resources. Nodes in the clusters had minimum specifications of single or duo core Intel X86 machines with 2.33 GHz processor speed, 4G memory and running Red Hat Linux. The experiments were conducted on 5-node clusters with block size set to 256MB. Scalability testing was done on clusters with 10, 15, 20, and 25 nodes. Pig release 0.7.0 and Hadoop 0.20 were used. All results recorded were averaged over three trials.

Testbed - Dataset and Queries: Synthetic datasets (n-triple format) generated using the BSBM tool were used. A comparative evaluation was carried out based on size of data ranging from 8.6GB (approx. 35 million triples) at the lower end, to a data size of 40GB (approx. 175 million triples). 10 queries (shown in Table 2.1) adapted from the BSBM benchmark (Explore use case) with at least a star and chain join were used. The evaluation tested the effect of query structure on performance with, (i) $Q1$ to $Q8$ consisting of two star patterns with varying cardinality, (ii) $2S1C$ consisting of two star patterns, a chain join, and a filter component (6 triple patterns), and (ii) $3S2C$ consisting of three star patterns, two chain joins, and a filter component (10 triple patterns).

2.4.2 Scalability of TripleGroup-based approach with size of RDF graphs

Figure 2.13 (a) shows the execution times of the three approaches on a 5-node cluster for $2S1C$. For all the four data sizes, we see a good percentage improvement in the execution times for RAPID+. The two star patterns in $2S1C$ are computed in two separate MR cycles in both the Pig approaches, resulting in the query compiling into a total of three MR cycles. However, RAPID+ benefits by the grouping-based join algorithm (`StarGroupFilter` operator) that computes the star patterns in a single MR cycle, thus reducing one MR cycle in total.

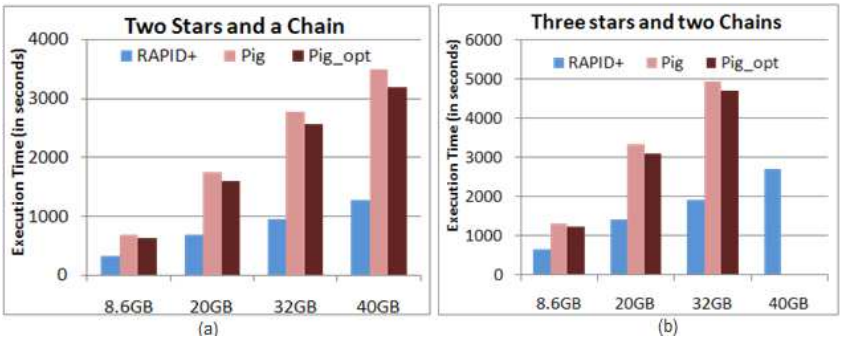


Figure 2.13: Cost analysis on 5-node cluster for (a) 2S1C (b) 3S2C

We also observe cost savings due to the integration of `loadFilter` operator in RAPID+ that coalesces the `LOAD` and `FILTER` phases. As expected, the Pig_{opt} performs better than the naive Pig approach due to the decrease in the size of the intermediate results.

Figure 2.13 (b) shows the performance comparison of the three approaches on a 5-node cluster for $3S2C$. This query compiles into three MR cycles in RAPID+ and five MR cycles in Pig / Pig_{opt}. We see similar results with RAPID+ outperforming the Pig based approaches, achieving up to 60% performance gain with the 32GB dataset. The Pig based approaches did not complete execution for the input data size of 40GB. We suspect that this was due to the large sizes of intermediate results. In this situation, the compact representation format offered by the RDFMap proved advantageous to the RAPID+ approach. In the current implementation, RAPID+ has the overhead that the computation of the star patterns results in a single relation containing TripleGroups belonging to different equivalence classes. In our future work, we will investigate techniques for delineating different types of intermediate results.

2.4.3 Scalability of TripleGroup-based pattern matching with denser star patterns

Table 2.1 summarizes the performance of RAPID+ and Pig for star-join queries with varying edges in each star sub graph. NTGA operators achieve a performance gain of 47% with $Q2$ (2:2 cardinality) which increases with denser star patterns, reaching 59% with $Q8$ (6:4 cardinality). In addition to the savings in MR cycle in RAPID+, this demonstrates the cost savings due to smaller intermediate relations achieved by eliminating redundant subject values and join triples that are no longer required. Figure 2.14 (b) shows a comparison on a 5-node cluster (20GB data size) with Pig_{opt} which eliminates join column redundancy in Pig, similar to RDFMap's concise representation of subjects within a TripleGroup. RAPID+ maintains a consistent performance gain of 50% across the varying density of the two star patterns.

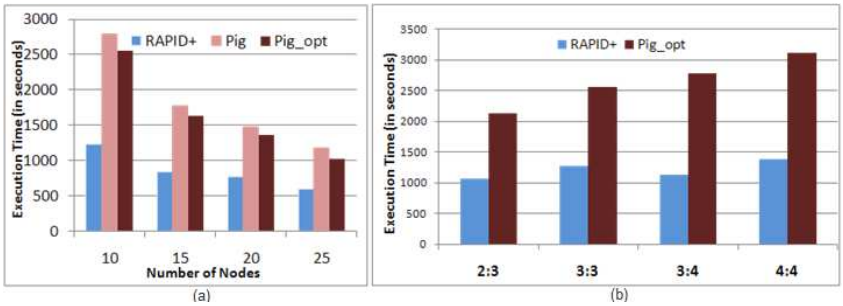


Figure 2.14: Scalability study for (a) 3S2C varying cluster sizes (b) two stars with varying cardinality

2.4.4 Scalability of NTGA operators with increasing cluster sizes

Figure 2.14(a) shows the scalability study of $3S2C$ on different sized clusters, for 32GB data. RAPID+ starts with a performance gain of about 56% with the 10-node cluster, but its advantage over Pig and Pig_{opt} reduces with increasing number of nodes. The increase in the number of nodes, decreases the size of data processed by each node, therefore reducing the probability of disk spills with the `SPLIT` operator in the Pig based approaches. However, RAPID+ still consistently outperforms the Pig based approaches with at least 45% performance gain in all experiments.

2.5 Chapter Summary

In this chapter, we discuss the issues and challenges in relational-style processing of basic graph pattern queries on MapReduce-based systems. We present an approach to evaluate graph pattern queries using an alternative algebra called the *Nested TripleGroup Data Model and Algebra* (NTGA). The NTGA-based approach concurrently executes "star joins", thereby enabling sharing of scans and computations across evaluation of multiple star patterns in a query. An implication of this is that NTGA-based query plans result in shortened MapReduce execution workflow, with reduction in I/O and network transfer costs.

Chapter 3

Nesting Strategies for Efficient Management of Intermediate Data

3.1 Motivation

Many real-world datasets contain multi-valued attributes or relationships e.g. friendships in a social network, citation references. An issue with this in join-intensive processing is that many of the combinations of tuples generated by a join operation contain some redundancy. Specifically, the subtuple containing the non-multi-valued attributes is repeated for each distinct value of the multi-valued attribute. For example, consider the join `SJ1` in Figure 3.1 which is a star-join between relations T_{pLabel}, T_{pProp}, and $T_{prodFeature}$ on the Sub column, to reassemble the label, property, and feature of products. Note that $prodFetaure$ is a multi-valued property that defines the one-to-many relationship between a product and its features. Consider the output tuples of the star-join in Out_{MR1} (Figure 3.1(a)) that represent details about a product $Prod1$ which has multiple product features ($PF1$, $PF2$ etc.). The subtuple labeled ($Sub1, Prop1, Obj1, Prop2, Obj2, Prop3$) is repeating for each distinct value of the product feature. We define redundancy factor of an output as the portion of redundant data in the output, that is written onto the HDFS at the end of a MapReduce cycle. Typically, the redundancy factor is proportional to the multiplicity of the multi-valued attribute. Muli-valued properties with high multiplicity such as Facebook friends of highly social persons, result in a high redundancy factor in intermediates when part of graph pattern queries.

Further, the redundancy factor is likely to compound across subsequent join operations i.e. the portion of redundant data in Out_{MR1} increases after the join `J1'` (refer to Out_{MR3} in Figure 3.1(b)). This ripple effect of redundancy in intermediate results has a negative impact on the the HDFS writes of the current cycle, as well as the HDFS reads and data shuffling costs of subsequent cycles. Figure 3.2 shows the amount of HDFS writes for a base query Q1 with two

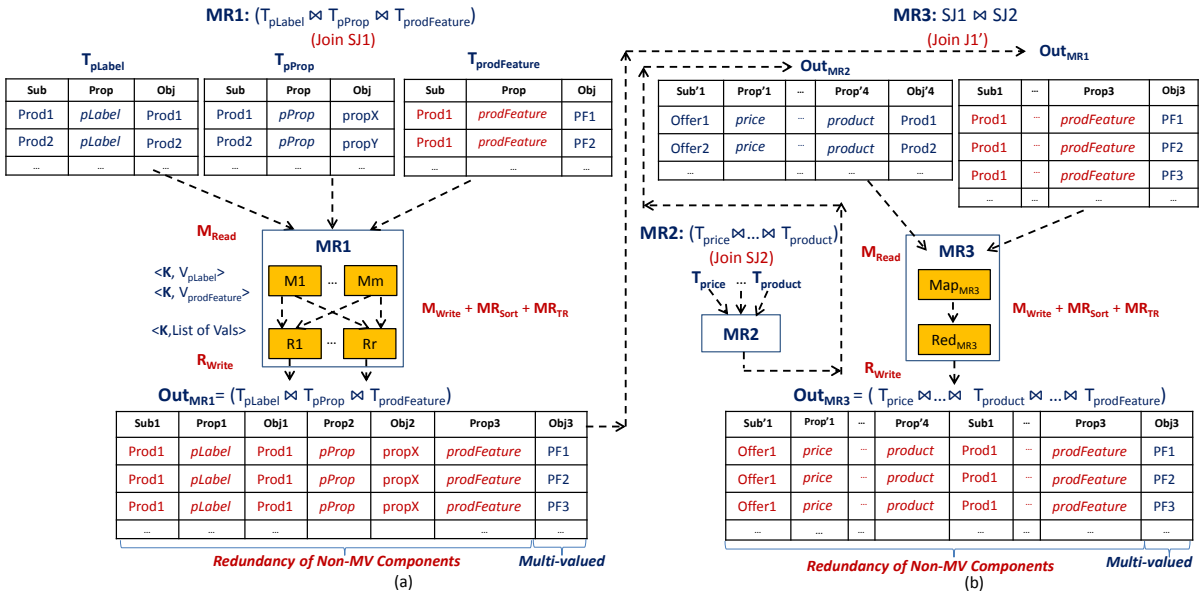


Figure 3.1: Ripple effect of the redundancy factor while processing flat intermediate relations with multi-valued attributes (a) intermediate relation Out_{MR1} containing a multi-valued attribute $prodFeature$ and repeated values for the non-multi-valued attributes (b) state blow-up in Out_{MR1} affects the costs ($M_{Read} + M_{Write} + MR_{Sort} + MR_{TR} + R_{Write}$) for the subsequent join cycle $MR3$ and also causes a ripple effect of the redundancy factor in the output relation Out_{MR3}

star-joins and no multi-valued (MV) attribute, and its variants Q2 and Q3 containing 1 MV attribute with low (6) and high (19) multiplicity respectively. The impact of the redundancy in intermediate results on the HDFS writes is significant while using flat data models. Hence, efficient management of redundancy while processing join-intensive data processing workloads is important to keep MR workflows nimble and cost-effective.

3.1.1 Related Work

There have been different techniques proposed recently for optimizing MapReduce data processing workflows. While some efforts [8, 10, 46, 77, 102] shorten the length of workflows to minimize the overall costs of MapReduce-based processing, some other efforts focus on sharing scans [68, 69, 99] and computations [68] across MapReduce workflows. The heuristic cost-based plan generator in [46] greedily groups the non-conflicting joins in a query to minimize the required number of MR cycles. HadoopDB [8] uses a hybrid database-Hadoop architecture to split execution between the database systems and Hadoop. However, the reduction in the number of MR cycles is limited to the portion of the query evaluation that can be pushed into the database, and is currently dependent on a tunable parameter [45] that determines the partitioning scheme.

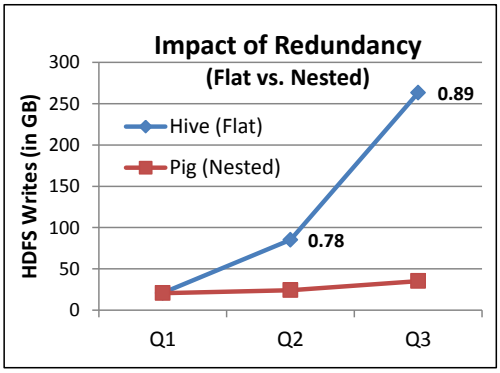


Figure 3.2: Impact of the redundancy factor in intermediate data on the amount of HDFS writes for queries Q1 (0 MV attribute), Q2 (1 low-multiplicity MV), and Q3 (1 high-multiplicity MV), and the benefit of using a nested model

Techniques that follow replicated join scheme [10, 102] to shorten the workflow, incur high data shuffling costs which is likely to worsen in the presence of multi-valued attributes.

Other techniques have been proposed to minimize the map output as well as the associated shuffle costs. Combiner() functions can be used to partially aggregate the map output locally at each mapper. In the case of different workflows that involve similar grouping operations on a common input relation, the sharing framework proposed in [68] merges them into a single workflow and enables sharing of map output data. Though this technique reduces the intermediate mapper-to-reducer transfer costs, details on extending this sharing mechanism to join operations is not provided. In [63], a value partitioning scheme is applied to manage potentially large and reducer-unfriendly groups during the cube computation process. In [97], a reducer routing strategy is supported that groups map keys in order to balance the data across reducers. Restore [38] stores the intermediate results of complete or partial MR workflows and reuses them subsequently, thereby reducing the overall execution time for subsequent queries. Sailfish [75] proposes an extension to the distributed file system to enable batching of disk I/O so as to optimize the data transfer between the mappers and the reducers.

In traditional database, the nested relational model [7] and object-oriented databases support sets, lists, maps and objects to concisely represent multi-valued attributes and complex objects with minimum data redundancy. Such extended models eliminate frequently occurring joins between objects by allowing nesting of related objects. Set-valued joins [44, 62] and other set-level comparisons [59] have been studied in the context of main-memory and centralized systems.

3.1.2 Contributions

This work proposes an approach that exploits the conciseness of a nested data model (NTGA) for avoiding redundancy in intermediate results that arises when data contains multi-valued attributes. NTGA has the advantage that its interpretation of subgraph pattern matching queries enables much shorter data processing workflows when compared to execution workflows produced by dataflow languages in Apache Hive and Pig which interpret such queries purely as join operations. Specifically, in this work, we extend the operators in NTGA with "nesting-awareness" [78,81] in a way that is MapReduce cognizant. This is in contrast to existing techniques that offer nested data models but no nesting-awareness in their operators. The consequence of the latter is that unnesting of intermediate data is often required to be completed in an earlier MR cycle prior to processing subsequent operation on the nested column. This work considers unnesting possibilities in either phase of a cycle, and discusses approaches for integrating the proposed ideas into the Apache Pig platform. The evaluation study compares the performance of our extended Apache Pig platform, the normal Pig processing framework and Apache Hive using both synthetic Semantic Web benchmarks and real-world data sets. The results demonstrate the benefits of the proposed strategies.

3.2 Intermediate Data Management in Complex MapReduce Workflows

The intermediate data flowing through a MapReduce data processing workflow impacts the initial data reads in the map phase (M_{Read}), the data shuffling costs that involve local disk writes at the mappers (M_{Write}), sort-merge costs (MR_{Sort}) as well as network transfer costs (MR_{TR}), and finally the cost of writing the reduce output to the HDFS (R_{Write}). For most tasks, the I/O and network transfer costs dominate the processing of the map and reduce functions. Hence, the cost of a MapReduce cycle MR_i can be summarized as:

$$M_{Read} + (M_{Write} + MR_{Sort} + MR_{TR}) + R_{Write}$$

Given that output of a MR cycle is read in by subsequent MR cycles, the overhead associated with bloated intermediate results could ripple across the workflow. In general, for successful completion of a workflow with k MR cycles MR_1 to MR_k, the amount of available disk space should be at least equal to:

$$(|Inp| + |Out_{MR_1}| + |Out_{MR_2}| + ... + |Out_{MR_k}|) \times Rep_{DFS}$$

where Inp is the initial input, Out_{MR_i} is reduce output for the ith MR cycle, and Rep_{DFS} is the configured replication factor of the distributed file system. In order to reduce the demand on disk space, we propose to capture , model and manage the redundancy in intermediate results.

3.2.1 Redundancy Factor due to Multi-valued Properties

The degree of redundancy or the redundancy factor in an intermediate relation is affected by several factors such as the degree of multiplicity of the multi-valued (MV) attribute, the size of the joining relation as well as the selectivity of the join. Without loss of generality, we formalize the concept of redundancy factor of a result in terms of results on 3-ary relations which capture Semantic Web graphs $(node, edge, node)$ or $(Subject, Property, Object)$ in Semantic Web parlance. Basically, a *Subject* and *Object* refers to nodes in a graph and *Property* is a labeled edge connecting the nodes and represents a typed relationship between the nodes. The property may be an attribute e.g. name, in which case the Object is a literal node representing a value. We note that queries with a filter on the MV column such as graph pattern queries with bound property-object pairs, are not likely to cause redundancy in intermediate results. Hence, for this discussion we focus on queries whose final result includes the multi-valued attribute.

Impact of redundancy on I/O costs

Let T_{P_j} be 3-ary (triple) relation containing instances of a particular labeled edge (Property type P_j) e.g. the set of names and their associated subject nodes. Let P_i be a multi-valued property with multiplicity M i.e. a subject node may have up to M distinct values for property type P_i. Consider a set of relational joins $(T_{P_1} \bowtie_s T_{P_2} \bowtie_s ... \bowtie_s T_{P_{i-1}}) \bowtie_s T_{P_i}$ (on the subject column) that is computed in the kth MapReduce cycle MR_k of an execution workflow. Let the output of MR_k be denoted as Out_k, containing tuples of the form $(Col_{11}, Col_{12}, Col_{13}, ..., Col_{i1}, Col_{i2}, Col_{i3})$ such that $Col_{x1}, Col_{x2}, Col_{x3}$ denote the subject, property, and object columns of the parent relation T_{P_x}. Then, the resultant join tuples in Out_k can be partitioned into f equivalence classes $C_1, C_2, .., C_f$ such that all tuples in a partition C_l agree on the non-MV components i.e. $(Col_{11}, ..., Col_{(i-1)3})$ and are therefore redundant. Further, since the multiplicity of P_i is M, the maximum number of tuples in any partition C_l is M. For example, if the number of distinct Products is 100, then the join result of `SJ1` in Figure 3.1(a) would be partitioned into 100 equivalence classes. All the tuples corresponding to product $Prod1$ agree on the non-MV columns (subject, property, object of all joining relations except for $T_{prodFeature}$), and hence belong to the same partition. Then, the amount of redundant data RD_k written to the disk during MR_k can be estimated as follows:

$$RD_k = \sum_{l=1}^{f} \sum_{\substack{t \in C_l \\ |C_l|>1}} \sum_{\substack{1 \le x \le i-1 \\ 1 \le y \le 3}} \mathrm{b}(t[Col_{xy}])$$

where $\mathrm{b}(t[Col_{xy}])$ is the size of column Col_{xy}. Thus, RD_k represents the size of the redundant columns that arise due to the presence of a multi-valued property in the current join operation. We define the ***redundancy factor*** $RedF_k$ for cycle MR_k as:

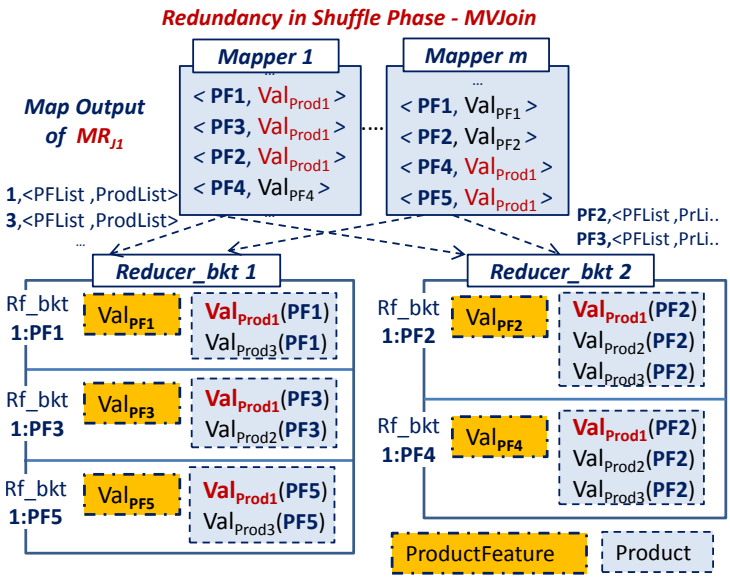


Figure 3.3: Snapshot of map and reduce phase for join J1 showing multiple copies of Val_{Prod1} being shipped to the same reducer node

$$RedF_k = \frac{RD_k}{|Out_k|}$$

where $|Out_k|$ represents the total size of the output Out_k. Thus, $RedF_k$ represents the proportion of the redundant data that is written into the HDFS at the end of cycle MR_k. The redundancy factor of a subsequent cycle MR_m ($RedF_m$) that reads the output of MR_k, is a function of ($RedF_k$*Sel_m), where Sel_m is the selectivity (Sel_m) of the operations processed in MR_m.

Impact of redundancy on network data transfer costs

Intermediate relations containing a multi-valued property may participate in subsequent join operations either based on the multi-valued column ($MVJoin$) or based on some other single-valued column ($non\text{-}MV\ join$). For the case of $MVJoin$, assume that the following tuples $(Prod1, pLabel, ..., prodFeature, PF1)$, $(Prod1, pLabel, ..., prodFeature, PF2)$ etc. (from Out_{MR1} in Figure 3.1(a)) need to be joined on the last column ($PF1$, $PF2$ etc.). Let Val_{Prod1} represent the subtuple consisting of the non-MV components of $Prod1$ i.e.

$$Val_{Prod1} = (Prod1, pLabel, Prod1, pProp, propX, prodFeature)$$

Then, in the $MVJoin$ phase, the mappers tag the tuples based on the join column (map output key = $PF1$ or $PF2$ etc.), and the non-MV subtuple forms the value (map output value = Val_{Prod1}). Figure 3.3 shows a snapshot of the map and reduce phase for this join operation. The map output tuples are partitioned and assigned to reducers (denoted as $Reducer\_bkt$),

where the reduce function is invoked on all tuples with the same key (denoted as Rf_bkt). For example, all tuples corresponding to the key $PF1$ are assigned to the reducer $Reducer\_bkt$ 1 and processed by the same reduce function Rf_bkt 1:$PF1$. We see that 5 copies of Val_{Prod1} (one each for values $PF1$, $PF2$, $PF3$, $PF4$, $PF5$) are processed at the mappers and shipped across to the reducers. Further, observe that 3 copies of Val_{Prod1} are shipped to the same reducer (denoted as $Reducer\_bkt$ 1). In this case, the entire value component of the map output is redundant and we refer to this as *full-value redundancy* in the shuffle phase. In the case of *non-MV join*, some portion of the value component (except the MV column) is redundant and so has a *partial-value redundancy* in the shuffle phase.

The data shuffling costs can be reduced by eliminating this redundancy. However, all tuples containing the value Val_{Prod1} are not guaranteed to be processed by the same mapper. So it may not be possible to address this redundancy issue in the map phase. Per-mapper shuffle redundancy can be eliminated if the reference to Val_{Prod1} can be shared across a mapper node - (i) each mapper tracks the processed tuples and groups them based on their value portion (as opposed to combining values based on keys in $Combiner$). Thus, tuples containing Val_{Prod1} are grouped together and their reference can be shared, (ii) a special partitioner then routes the map output tuples based on a group key ($PF1$, $PF3$, $PF5$ map to the same group key since they are assigned to the same reducer), and (iii) a meta-reduce function clones Val_{Prod1} at the reducer and ships them to the appropriate reduce functions based on their original keys ($PF1$, $PF2$ etc.). The overhead of steps (i) (comparing value of each tuple) and (iii) (reduce-side cloning of tuples) may be significant based on the multiplicity of the multi-valued property. Also, this approach ensures that at most 1 copy of Val_{Prod1} is shipped from each mapper to each reducer, which may still be significant for cycles with large number of mappers and reducers. However, if all the tuples containing Val_{Prod1} were processed by a single mapper, we could enable more efficient reference sharing that minimizes the redundancy in the shuffle phase. The rest of the chapter focuses on nesting and unnesting strategies in the context of some previous work on a nested data model and algebra (NTGA).

3.2.2 Minimizing Redundancy in Intermediate Results using NTGA's Nested Data Model

As described earlier, NTGA-based execution plans require less numbers of MR cycles when compared to its relational counterpart. Our example query in Figure 3.1 can be executed in a total of 2 MR cycles using the NTGA approach as opposed to 3 MR cycles using the relational approach. In addition to minimizing the number of required MR cycles, NTGA's nested data model allows concise representation of intermediate results and avoids redundancy in the case of multi-valued properties. Figure 3.4 represents an example TripleGroup corresponding to a

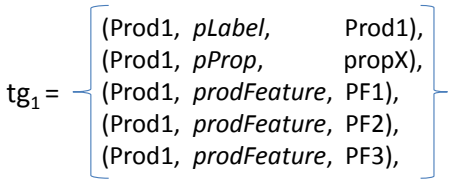


Figure 3.4: An example triplegroup implicitly representing star subgraphs containing a multi-valued property *prodFeature*

star subgraph that is a valid result for the star-join $SJ1$ in Figure 3.1(a). Note that n-tuples containing multi-valued properties such as

(Prod1, *pLabel*, Prod1, *pProp*, propX, *prodFeature*, PF1)
(Prod1, *pLabel*, Prod1, *pProp*, propX, *prodFeature*, PF2)
(Prod1, *pLabel*, Prod1, *pProp*, propX, *prodFeature*, PF3)

are implicitly represented using a single triplegroup tg_1 in Figure 3.4. This avoids the redundancy in intermediate results, and thus reduces the amount of HDFS writes at the end of the reduce of the $SJ1$ MR cycle. The next section builds on the advantages of using a nested model in efficient management of intermediate results while processing queries with multi-valued properties.

3.3 Unnesting Strategies for Efficient Management of Multi-valued Properties

Given a nested data model, there are three possible unnesting strategies that can be considered for dealing with redundancy in intermediate results, (i) *early complete unnesting* in the reduce phase of the cycle that generates the input to the *MVJoin* operation, (ii) *lazy complete unnesting*, and (iii) *lazy partial unnesting*. Both (ii) and (iii) are processed in the map phase of the cycle processing the *MVJoin* operation.

Figure 4.5 represents an overview of the three strategies and their corresponding query plans for an example query with two star-join operations (`SJ1` and `SJ2`), and a third join `J1` which is an *MVJoin* operation (join on the MV column). Figure 4.5 denotes the stages at which the intermediate results containing an MV property are nested and unnested (flattened). For the rest of this discussion, we use *Star-MVP* to denote a star subgraph containing an MV property (such as `SJ1` in Figure 3.1). While the first strategy in Figure 4.5(a) is based on the Pig approach, the other two are NTGA-based plans, which we elaborate next.

3.3.1 Early Complete Unnesting: Reduce-side Full Replication

Apache Pig's nested data model can be exploited to eliminate data redundancy while representing star-join results corresponding to Star-MVP. This can be achieved by processing star-joins

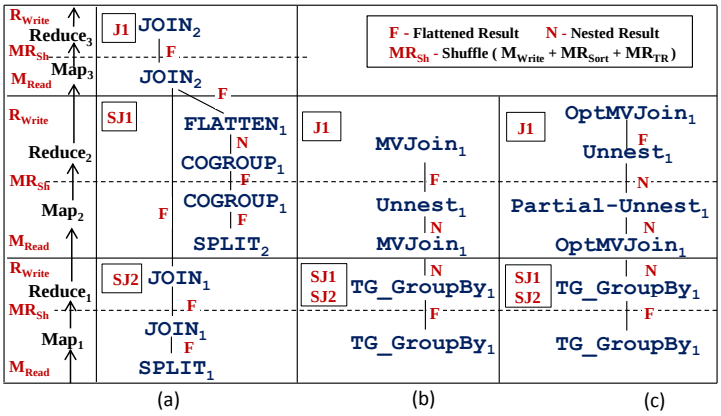


Figure 3.5: Unnesting Strategies for *MVJoin* (a) *early complete* unnesting in reduce of MR_{SJ1} (b) *lazy complete* unnesting and (c) *lazy partial* unnesting in map of MR_{J1}

as a `COGROUP` operation that is used to group multiple relations on the same column, such as the Subject column in this case.

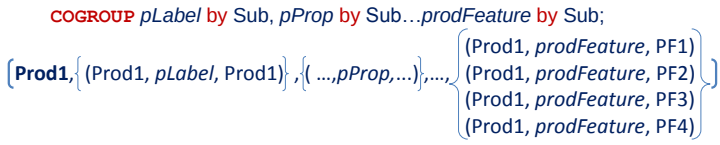


Figure 3.6: Nested tuple from the COGROUP operation on vertically partitioned relations

A `COGROUP` on N relations, results in a nested tuple with N columns, where each column is a bag containing corresponding tuples from the participating relations as represented in Figure 3.6. The `COGROUP` based approach for star-join computation of Star-MVPs has no redundant non-MV components, and hence minimizes the redundancy factor for MR_{SJ1} ($RedF_{SJ1}$=0). The reduced $RedF_{SJ1}$ results in small amount of disk writes (R_{Write}) in MR_{SJ1}. However, each 'column' in the result of a `COGROUP` is a bag of tuples, and Pigs `JOIN` operator is not defined on nested columns. Hence, processing any subsequent join operation requires unnesting (or flattening in Pig Latin parlance) of the join column.

In the case of a join operation on any of the non-MV components such as object of *pLabel*, we may *partially* unnest the nested tuple i.e. unnest only the non-MV column, which does not increase the resultant tuples and hence does not affect $RedF_{SJ1}$. However, the case of an *MVJoin* requires complete unnesting of the tuples based on the MV column prior to any subsequent join operation, resulting in redundant non-MV components in the intermediate tuples, similar to the redundant component in Out_{MR1} in Figure 3.1(a). This unnesting would happen in the reduce

of the phase that generates the input to the *MVJoin* operation. Both partial and complete unnesting can be achieved using the `FLATTEN` operator at the end of the reduce phase of the `COGROUP` as shown in Figure 4.5(a). The complete unnesting of the nested tuple results in full replication of the Star-MVP i.e. the replication factor *Rep* is a function of the multiplicity of the multi-valued property.

3.3.2 Lazy Complete Unnesting: Map-side Full Replication

NTGA operators are nesting-aware and do not require unnesting before operations such as join. This allows the unnesting of Star-MVPs to be delayed to the map phase of the *MVJoin* operation as opposed to the reduce phase of a previous cycle. For example, the unnesting of Star-MVP can be delayed till MR_{J1} using NTGA (Figure 4.5(b)), as opposed to unnesting in MR_{SJ1} using the Pig approach (Figure 4.5(a)).

RAPID+ uses an internal representation scheme, *RDFMap* (extended multi-map) that concisely represents subject-triplegroups and supports efficient look-up of (*Property*, *Object*) pairs. In order to support multi-valued properties, RDFMap is extended to support (*Property*, List$\langle$*Object*$\rangle$) pairs such as (*prodFeature*, $\{PF1, PF2, PF3, PF4, PF5\}$) as shown in Figure 3.7 (top). For the rest of this discussion, we use the notation $rMap_{Prod1}(PF1, ..., PFn)$ to refer to a triplegroup corresponding to Product $Prod1$ and containing an MV property *prodFeature* with n object values $PF1, ..., PFn$.

In the lazy unnesting strategies for graph pattern queries involving *MVJoin*, the unnesting of $rMap_{Prod1}$ is delayed till the map phase of the *MVJoin* operation. The map-side `unnest` operation is integrated into the map phase of NTGA's join, which completely unnests a Star-MVP and generates a map output tuple for each flattened copy of the Star-MVP. For example, $rMap_{Prod1}$ is unnested into 5 triplegroups, one for each of the distinct product features. Thus, the replication factor *Rep* is a function of the multiplicity of the multi-valued property. The lazy unnesting approach minimizes the redundancy factor in the output of MR_{SJ1} ($RedF_{SJ1}$=0) resulting in savings in disk writes (R_{Write}) in the star-join phase as well as reduced amount of reads (M_{Read}) in the subsequent *MVJoin* phase.

3.3.3 Lazy Partial Unnesting: Map-side Partial Replication

If the multiplicity of a multi-valued property is greater than the number of partitions in the reducer space, it is likely that multiple copies of the Star-MVP are assigned to the same partition. Let us revisit Figure 3.3 with 2 reducers (r=2), where 3 copies of the map output value Val_{Prod1} corresponding to the join keys $PF1$, $PF3$, $PF5$ respectively, are mapped to the same *Reducer_bkt* 1. The map-side sorting costs (MR_{Sort}), local writes (M_{Write}) and network communication costs (MR_{TR}) can be reduced if the references to Val_{Prod1} can be shared across

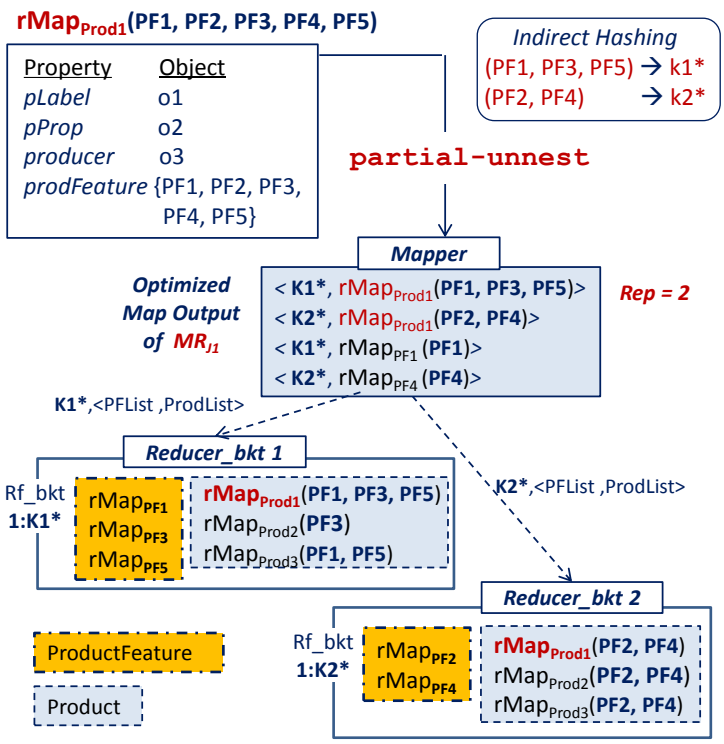


Figure 3.7: Lazy map-side partial unnesting ($Rep = 2$)

the reduce function space (rf_bkt or a group of tuples processed by the same reduce function) i.e., if the replication factor Rep can be reduced.

We propose an alternative key-assignment strategy for MVJoins, such that multiple map keys can be assigned to the same rf_bkt in the reduce function space. This partitioning scheme is based on *indirect hashing* of the map output key to the Reducer space. We define a function $func^*$ that is used to map intermediate keys to the Reducer space. Given an intermediate key $k1$ we define:

$$func^*(k1) = k1^*$$

that maps the original key $k1$ to a group key $k1^*$. The default partitioning scheme is then applied based on the group key $k1^*$:

$$\text{partition}(k1) = \text{hash}(\ k1^*\ )\ \%\ r$$

In effect, an intermediate key-value pair ($k1$,$v1$) is assigned to $Reducer\_bkt_{hash(k1*)\%r}$. Any two intermediate keys $k1$ and $k2$ are assigned to the same rf_bkt if $k1^* = k2^*$. Consider the map phase of the MVJoin operation with an example map input $rMap_{Prod1}$ as shown in Figure 3.7. Let us assume that $func(PF1)^*$=$func(PF3)^*$=$func(PF5)*$. Then, the keys $PF1$, $PF3$, $PF5$ map to the same group key and hence are assigned to the same bucket rf_bkt

(1 : $K1$*). Consequently, only 1 copy of $rMap_{Prod1}$ is transferred to *Reducer_bkt* 1, reducing the shuffle costs ($MR_{Sort} + M_{Write} + MR_{TR}$). The `partial-unnest` operation partially unnests the triplegroup based on the grouping function $func$*, and is integrated into the map phase of NTGA's join operator.

Algorithm 3: OptMVJoin (Optimized MVJoin)

```
   OptMap (key:null, val: RDFMap rMap) ;
1  if join on Sub then
2      emit ⟨ func*(rMap.Sub), rMap⟩ ;
   else if join on Obj then
       //Partially unnest MVP's ObjList in rMap based on k*=func*(Obj)
3      pList ← partial-unnest (rMap, func*) ;
4      foreach partialMap ∈ pList do
5          key ← extract k* for partialMap ;
6          emit ⟨ key, partialMap⟩ ;

   OptReduce (key:k*, val:List of RDFMaps R) ;
7  leftList ← extract leftEC RDFMaps from R //MV
8  rightHash ← extract rightEC RDFMaps from R //non-MV
9  foreach leftR ∈ leftList do
       //Handle multi-valued property
10     MVList ← extract prop's objList from leftR ;
11     foreach joinKey ∈ MVList do
12         rightR ← rightHash.get(joinKey) ;
13         emit ⟨ joinRDFMaps(leftR, rightR) ⟩ ;

   partial-unnest (RDFMap rMap, func*) ;
   //RDFMap(Sub, EC, propMap)
14 objList ← extract MV prop's list of objects from propMap;
15 foreach obj ∈ objList do
16     groupKey ← func*(obj) ;
17     add (prop, obj) to partialList[groupKey];
18 emit partialList;
```

Algorithm 3 shows the map and reduce functions for the `OptMVJoin` operator (optimized TG_{Join} for joins involving multi-valued property). In the map phase, the RDFMaps that join on subject Sub are annotated using its group key k^* computed using $k^* = func^*(Sub)$ (lines 1-2). For joins on object, the RDFMap is partially unnested using the `partial-unnest` operation. The `partial-unnest` operator splits the object list of the multi-valued property based on the Object's group key $k* = func*(Obj)$, resulting in a list of partially-unnested RDFMaps (*pList* in line 3). A map output tuple is generated for each partially-unnested RDFMap, annotated by its group key (lines 4-6). The replication factor *Rep* is now a function of $func^*$. In the reduce phase, all RDFMaps corresponding to the same group key k^* but different join keys are processed in the same reduce(). For example, the RDFMaps in *Reducer_bkt*1 correspond to the group key $k1*$ but different original keys $PF1$, $PF3$, and $PF5$. This requires selectively joining the

RDFMaps based on the original join key. RDFMaps corresponding to the left relation ($leftEC$) are extracted into a list (line 7). RDFMaps from the right relation ($rightEC$) are unnested and hashed based on the join key (line 8). The algorithm iterates through each RDFMap in the left relation (line 9), and probes the hashed relation for each distinct object value (join key) for each property (lines 10-12). When a match is found the two RDFMaps are joined (line 14) as per the definition of `TG_Join`.

3.3.4 Implementation Issues

One challenge with using nested data models is the issue of estimating appropriate number of mappers and reducers based on the potential size of the map output. By default, the number of mappers and reducers are initiated based on the size of the input. Nested representation of intermediate relations results in the initiation of small number of mappers / reducers. For certain queries, it was observed that Pig and Hive initiated 50% more mappers for the *MVJoin* phase when compared to the NTGA-based MR plan. This negatively impacts NTGA since the map-side unnesting of nested results, create a map output size that is larger than the map input. Better parallelization can be achieved if the numbers of mappers and reducers can be adjusted accordingly.

Estimating Number of Mappers for MVJoin. By default, the estimation of the number of mappers is guided by the input size and the HDFS block size. More number of mappers can be forced by (i) using a smaller block size, (ii) setting the `setCombineSmallSplits` parameter to false which avoids combining smaller output files from the previous reduce cycle, and (iii) increasing the number of reducers used in the previous MR cycle. By (ii) we can ensure that the number of mappers initiated will be at least equal to the number of reducers used in the previous cycle. For our evaluation, we used a combination of (ii) and (iii). For the star-join cycle, instead of estimating the number of reducers based on the input, we rounded it off to the maximum number of reduce slots available i.e. for a 10-node cluster, we used the number of reducers for the star-join cycle to be 20. For this example, the number of mappers for the *MVJoin* will at the least be 20.

Estimating Number of Reducers for MVJoin. By default, Pig estimates the number of reducers based on the size of the input data as (total size of input data) / (number of input bytes per reducer). The number of bytes per reducer can be set using `pig.exec.reducers.bytes.per.reduce` (default being 1GB). In order to accommodate the blow-up of the map output due to unnesting at the map phase, we estimate the potential size of the map output based on the average multiplicity of the MV property and the average size of the non-MV component. Assume that the *MVJoin* phase computes the join between star subpatterns `SJ1` and `SJ2`. Let $|TP_{SJ1}|$ and $|TP_{SJ2}|$ be the number of triple patterns in the star subpatterns `SJ1` and `SJ2`

respectively. Let $|TG_{SJ1}|$ and $|TG_{SJ2}|$ be the number of triplegroups corresponding to the star subpatterns `SJ1` and `SJ2` respectively. Assume that star subpattern `SJ2` contains a multi-valued property with average multiplicity M. Note that all these statistics can be collected during the initial star-join phase in NTGA-based processing. Then, the potential map output M_out^+ can be estimated as follows:

$$|M\_out^+| = |M\_inp| \times \frac{(|TP_{SJ1}| \times |TG_{SJ1}| + M \times |TP_{SJ2}| \times |TG_{SJ2}|)}{(|TP_{SJ1}| \times |TG_{SJ1}| + (|TP_{SJ2}| + \frac{M}{3}) \times |TG_{SJ2}|)}$$

where $|M\_inp|$ denotes the map input size (in bytes). We found by experience that it is beneficial to round off the estimated number of reducers to be a multiple of number of reducer nodes available.

Note on $func^*$ **.** The function $func^*$ is parameterized by a partition factor N, where N depends on several factors such as the potential size of the map output M_out^+, number of reducer nodes, and average number of tuples that can be processed by a reducer. The impact of some of these factors is discussed in the evaluation section. If the required number of reducers is r, then N must be equal to or greater than r to avoid heap errors. Ensuring that the value of N be a multiple of r helps with uniform distribution of map output keys i.e. all rf_bkt partitions will be of roughly equal size. However, if the data is highly skewed $func^*$ may need to be customized to avoid mapping two frequent keys $k1$ and $k2$ to the same group key.

3.4 Empirical Evaluation

The main goal of our evaluation was to study the impact of the proposed nesting and unnesting strategies on minimizing the redundancy factor in intermediate results while processing graph pattern queries. For relational-style systems, we evaluated Apache Pig (*Pig-Def*, *Pig-Opt*) and Hive (*Hive*), both of which support tuple-based algebra. We use *NTGA* and *NTGA-Opt* to denote the lazy unnesting strategies with map-side full replication and partial replication respectively.

3.4.1 Setup

Environment: The experiments were conducted on VCL [83] an on-demand virtual computing environment provided by NC State University. Each node in the cluster was a dual core Intel X86 machine with 2.33 GHz processor speed, 4G memory and running Red Hat Linux. The experiments were conducted on 5-node and 10-node Hadoop clusters with block size set to 256MB and heap-size for child jvms set to 1024MB. Pig release 0.10.0, Hive 0.8.1 and Hadoop 0.20.2 were used. All results recorded were averaged over two or more trials.

Table 3.1: Testbed graph pattern queries with multi-valued properties

Query	Joins	Edges in Star-patterns
low-1Star, high-1Star	3	$S_1{:}4_{1mvp}$
base-2Star	8	$S_1{:}5 \bowtie S_2{:}4$
low-2Star, high-2Star	8	$S_1{:}5 \bowtie S_2{:}4_{1mvp}$
MV-2p	4	$S_1{:}2_{1mvp} \bowtie S_2{:}3$
MV-3p	5	$S_1{:}3_{1mvp} \bowtie S_2{:}3$
MV-4p	6	$S_1{:}4_{1mvp} \bowtie S_2{:}3$
MV-5p	4	$S_1{:}5_{1mvp} \bowtie S_2{:}3$
Dbp1	4	$S_1{:}3_{2mvp} \bowtie S_2{:}2_{2mvp}$
Dbp2	5	$S_1{:}4_{2mvp} \bowtie S_2{:}2_{2mvp}$
Dbp3	7	$S_1{:}4_{2mvp} \bowtie S_2{:}4_{3mvp}$
Dbp4	7	$S_1{:}4_{3mvp} \bowtie S_2{:}4_{3mvp}$
Dbp5	8	$S_1{:}5_{4mvp} \bowtie S_2{:}4_{3mvp}$

Testbed - Dataset and Queries: Both synthetic and real-world datasets were used for evaluation. The Berlin SPARQL Benchmark (BSBM) [22] dataset includes two MV properties, both defined for a class of Products $productFeature$ with approximate multiplicity 19 and product *type* with multiplicity 6. The BSBM data generator was used to generate synthetic datasets in n-triple format. Three data sizes were generated using number of Products as the scalability factor *BSBM-250k*, *BSBM-500k*, *BSBM-1000k*, with the size of data ranging from 22GB (*BSBM-250k* with 250,000 Products and approx. 87M triples in total) to a data size of 87GB (*BSBM-1000k* with 1000,000 Products and approx. 350M triples in total). Two categories of queries that involve multi-valued properties were considered, (i) *non-MV join* - join variable is single-valued, and (ii) *MVJoin* - join variable is the object of a multi-valued property. The evaluated graph pattern query structures (star-join structures denoted as S_1, S_2, S_3) are represented in Table 3.1. The DBPedia Infobox (DbInfobox) [16] includes real-world data with 33.74M triples (20.5M properties and 13.23M types) of size 4.4GB. More than 45% of the properties in DbInfobox are multi-valued with varying multiplicity. For example, the multiplicity of the property $influenced$ varies from 1 to as high as 50 for certain instances. The DbInfobox queries chosen for evaluation have multiple multi-valued properties, each with varying multiplicity. Additional details about the evaluated queries, including the Pig and Hive scripts are available in Appendix B.

Note on the Approaches: Before we analyse the evaluation results we remind the readers about the difference in MapReduce workflows and the optimizations in the different approaches. *Hive* enables shared scan of input relation during n-way (star) join cycles while *Pig* does not. Hence, though *Pig* and *Hive* produce same length MR workflows for all the queries, *Pig* in general has higher HDFS reads and uses more number of mappers. The grouping-based approach in *NTGA* results in shorter execution workflows when compared to *Pig* / *Hive* and also enables shared scan of the input relation. *Pig-Def* and *Pig-Opt* use VP approach via Pig's

Table 3.2: Nesting and Unnesting Strategies

Approach	Star-join Phase (Multiplicity M)	MVJoin Phase
Hive *Pig-Def*	M tuples (no nesting)	–
Pig-Opt	$MVJoin$:M tuples Other: 1 nested tuple	–
NTGA	1 nested $RDFMap$	`map-side-full-unnest`
NTGA-Opt	1 nested $RDFMap$	`map-side-partial-unnest`

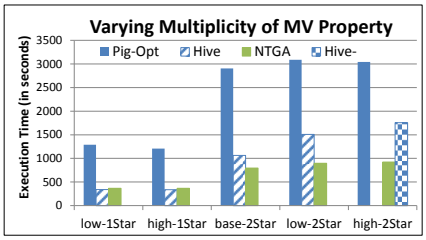
`SPLIT` operator. *Pig-Opt* uses `COGROUP` for non-MV joins (as described in section 3.3.1). Table 3.2 summarizes the nesting and unnesting strategies used in the evaluated approaches.

3.4.2 Impact of the Nesting Strategy

In this section, we evaluated queries where the redundancy factor can be completely eliminated using nested models i.e. *non-MV join* queries where all the join columns in the query are single-valued. We study the impact of the nesting strategy with the varying degree of redundancy factor, which depends on the multiplicity of the multi-valued property as well as the size of the joining relations.

Low to high multiplicity of MV properties: Figure 3.8(a) shows the performance evaluation of *Pig-Opt*, *Hive*, and *NTGA* for queries containing 1 multi-valued property with low and high multiplicity of 6 (Product Type) and 19 (Product Feature) respectively. In systems that use flat tuple-based algebra (Pig-Def / Hive), the star-join result containing a MV property with multiplicity M is represented as M tuples, and contains redundancy. Figure 3.8(b) denotes the redundancy factor in intermediate results while evaluating the queries using flat algebra. Queries *low-1Star* and *high-2Star* can be computed in a single MR cycle (MR_{S1}) and their reduce output contains a redundancy factor of 0.72 and 0.82 respectively, when evaluated using a Hive. The two star subpattern queries demonstrate how the redundancy factor compounds across the subsequent join cycle. While the redundancy factor of *low-2Star* increases from 0.72 (in MR_{S1}) to 0.78 after the subsequent join in $MR_{S1 \bowtie S2}$, for *high-2Star* it increases from 0.82 (in MR_{S1}) to 0.89 (in $MR_{S1 \bowtie S2}$).

Though this increase in redundancy factor seems insignificant, its impact on the HDFS writes can be seen in Figure 3.8(a). In fact, for the BSBM-500K dataset the *Hive*/*Pig-Def* approaches failed to complete execution for *high-2Star* on a 10-node cluster due to insufficient disk space (denoted as a missing bar for Hive in Figure 3.8(a)). This failure can be attributed to the blow-up of the intermediate results. *Hive* approach occupied 52% more disk space after the star-join phase when compared to the nested approaches. On the contrary, the nested approaches

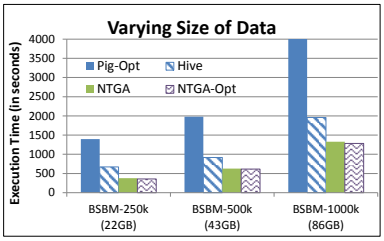


(a)

Redundancy Factor Across the MapReduce Workflow

Query	MR_{S1}	MR_{S2}	$MR_{S1 \bowtie S2}$
Low-1Star	0.72 (1.5GB)	-	-
High-1Star	0.82 (5.2GB)	-	-
Base-2Star	0 (0.2GB)	0 (7.7GB)	0 (12.7GB)
Low-2Star	0.72 (1.6GB)	0 (7.7GB)	0.78 (75.8GB)
High-2Star	0.82 (5.4GB)	0 (7.7GB)	0.89 (250GB)

(b)



(c)

Figure 3.8: Varying multiplicity of a multi-valued property (a) comparative evaluation using one and two star sub-pattern queries containing low and high multiplicity MV property (BSBM-500K, 10-node)(b) redundancy factor in reduce output while evaluating the given queries using flat algebra (c) scalability study of the nesting and lazy unnesting strategies with increasing size of data (MV-5p, 10-node)

(*Pig-Opt* and *NTGA*) required 71.5% / 86.6% less disk space overall, when compared to Hive for queries *low-1Star* / *high-2Star* respectively.

Varying number of star subpatterns: For clarity on the impact of the nesting strategy, we also include a baseline query *base-2Star* that has the same number of joins as *low-2Star* and *high-2Star* but does not contain any multi-valued property. For the one star subpattern query, all three approaches require just 1 MR cycle and the execution time for *Hive* and *NTGA* are comparable. For the two star subpattern queries, *NTGA* clearly outperforms the other two approaches. *NTGA* has a performance gain of 24% over *Hive* for the baseline query which increases up to 40% for *low-2Star*. Since *Hive* failed for *high-2Star*, we modified the query to project out property columns (denoted as *Hive-* in Figure 3.8(a)) so as to reduce the redundancy factor and the required disk space. *NTGA* (without any additional projections) still had a performance gain of up to 47% over *Hive-*. Further, *NTGA* had a performance gain of 70% over *Pig-Opt* for all the 3 two star subpattern queries.

3.4.3 Impact of Lazy Unnesting Strategies

Figure 3.9 shows a detailed performance comparison of the map and reduce execution times for *Hive*, *NTGA* and *NTGA-Opt* for query *Mv-5p* with two star subpatterns and a $MVJoin$. In *Hive*, the result of the star-joins are stored as flat n-tuples and the $MVJoin$ is processed using the default relational-style join algorithm ($MR3$ in Figure 3.9). Both *NTGA* and *NTGA-Opt* concisely represent the star-join result. *NTGA* uses the map-side complete unnesting and requires a more complex map phase during the $MVJoin$ cycle (MR2) to generate M copies of the RDFMap, one for each object value of the MV property. However, *NTGA-Opt*'s map phase for $MVJoin$ (MR2) integrates indirect hashing that results in lesser number of map output tuples when compared to the other two approaches. The references of map output are shared across reducer tasks assigned to the same reducer partition, resulting in reduced shuffle bytes,

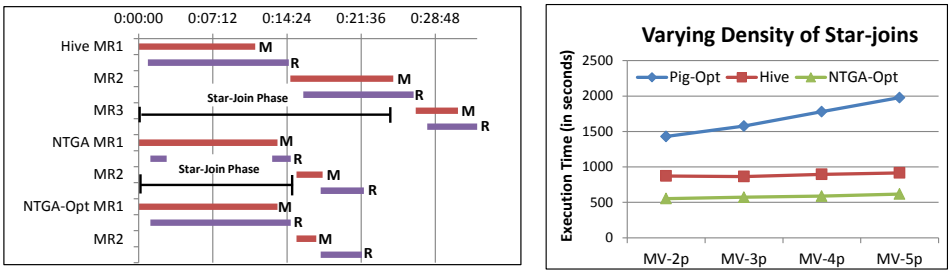


Figure 3.9: (a) A comparison of map and reduce execution times for MVJoin query MV-5p (BSBM-1000k, 10-node), (b) Impact of lazy unnesting strategy with increasing cardinality of star-joins (BSBM-500k, 10-node)

reduced costs of sort and data transfer.

Varying density of star-joins: We studied the impact of the density of the star-joins containing the multi-valued property by evaluating four different query patterns (MV-2p to MV-5p). Denser star-join structures result in larger size of non-MV components and hence a higher redundancy factor. The lazy unnesting strategies in *NTGA* outperform both *Hive* and the early complete unnesting in *Pig-Opt* for all the queries on BSBM-500k (10-node cluster) . As the size of the redundant component increases, *NTGA* shows an increasing performance gain over *Pig-Opt* from 61% in *MV-2p* to 68% in *MV-5p*.

Varying data size: Figure 3.8(c) shows a comparative evaluation of the four approaches with increasing size of RDF graphs, varying from 87M triples to 350M triples. The two star query *MV-5p* was evaluated on a 10-node cluster. The increasing data size impacted Pig and Hive approaches the most. Both the lazy unnesting strategies have an overall performance gain of 30-34% and 68-70% over Hive and Pig approaches respectively.

Varying partition factor (N): In this section, we study the impact of the partition factor N (a parameter to $func^*$) on the redundancy factor and the associated costs. The redundancy in the shuffle phase can be controlled by varying the partition factor N that is used to group the join keys. Table 3.3 shows the execution time for the $MVJoin$ phase (MR_3 in Pig-Def and MR_2 in NTGA) for query *MV-5p*, along with the impact of the varying reduce groups. Both Pig-Def and NTGA, use equal number of reduce groups based on the number of distinct join keys (47884). The *MV Rep* column denotes the replication factor in the map output of the *MVJoin* phase. For *Pig-Def* and *NTGA*, the MV Rep is equal to the multiplicity of the MV property i.e. 19.43. In general, a lower value of N maps more keys to the same group and hence enables better sharing of data references and reduces replication (MV Rep decreases to 13% for N=20). This results in lower shuffle costs, but increases the average time taken by a reduce task since large number of records are processed by each reduce function. On the other hand, a higher value of N provides less sharing opportunities. The cost model for estimation of N needs to take into account the size of data that can be processed by a reducer based on its heap

Table 3.3: Impact of varying partition factor on MVJoin phase (5-node, BSBM-250k, MV-5p)

Approach (Reduce Grps)	Map Output Records	MV Rep (%)	Map (s)	Reduce (s) (avg. shuffle)
Pig-Def(47884)	4.905M	19.43	179	246 (121)
NTGA(47884)	4.905M	19.43	329	288 (179)
NTGAOpt(1000)	4.873M	19.30	273	266 (159)
NTGAOpt(100)	4.583M	18.14	249	257 (144)
NTGAOpt(80)	4.483M	17.74	237	255 (139)
NTGAOpt(60)	4.246M	16.80	223	245 (130)
NTGAOpt(40)	4.064M	16.06	217	244 (126)
NTGAOpt(20)	3.326M	13.11	185	212 (95)

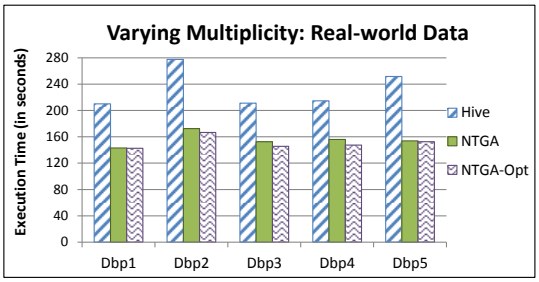


Figure 3.10: Impact of varying multiplicity of multi-valued properties (DbInfobox, 5-node)

size. Additional experiments on test queries *MVJoin-First* and *MVJoin-Last* demonstrated the impact of the join order in the selection of the partition factor.

Varying multiplicity in real-world data: Queries evaluated in this section consisted of multiple multi-valued properties, whose multiplicity varied across instances. Of the tuple-based approaches, we evaluate only Hive, since previous experiments across varying structures of graph pattern queries have clearly shown that Hive outperforms Pig. Figure 4.13 shows a comparison of execution times for *Hive* and the two lazy unnesting strategies. Overall, the lazy unnesting strategies show about 30%-40% performance gain over *Hive* for all queries. Queries *Dbp*4 and *Dbp*5 consist of the same number and multiplicity of multi-valued properties and the intermediate results of the initial join cycles have equal redundancy factor. However, the final join in *Dbp*5 has higher selectivity than that of *Dbp*4, and hence the redundant component blows up in the final join cycle. The average multiplicity of an MV property depends on (i) its defined multiplicity which can be either uniform (property type *name* is defined 1-2 times for all resources) or may vary across instances (property type $influenced$ is defined 2-3 times for some and up to 50 times for few resources), and (ii) the number of instances participating in the *MVJoin*.

3.5 Chapter Summary

In this chapter, we present an argument for *nesting-aware* dataflow operators for MapReduce-based data processing workflows, as a means of eliminating avoidable costs associated with redundancy in intermediate results. Using such operators, intermediate results can remain in their nested (non-redundant) form and be unnested only when required and at the latest possible time. We also present a discussion on how such nesting-aware operators and different *unnesting strategies* are integrated into the Apache Pig platform.

Chapter 4

Optimization of Unbound-property Graph Pattern Queries

4.1 Motivation

The successful adoption of Semantic Web technologies to interlink diverse (related) datasets has led to large semantically-integrated scientific (Uniprot [17], Bio2RDF [18]) and general purpose (DBpedia [16], Billion Triple Challenge [3]) RDF data warehouses. The evolving and heterogeneous nature of such data collections makes it difficult for users to be familiar with different kinds of relationships that exist in the data. Consequently, exploration of datasets in data-integration [67] and data archival [90] scenarios require flexibility in querying, i.e.,the ability to use structural variables or "don't cares" in queries. SPARQL [74], the standard query language to specify graph pattern queries on the Semantic Web, enables flexible querying of datasets by allowing `OPTIONAL` substructures or substructures with missing edge labels. The latter are called *unbound-property* triple patterns and can be used to query unknown relationships ("*Scientists in some way associated to the same city*"), relationships with partial knowledge ("*Gene Ontology terms related to a gene Rxr*"), or to retrieve all available information about a resource ("*What is known about the Hexokinase gene?*").

Consider an example SPARQL query $Q1$ on Bio2RDF, a Life Sciences RDF dataset. $Q1$ is useful to analyse Parkinson's disease and has two unbound-property triple patterns (1) and (5).

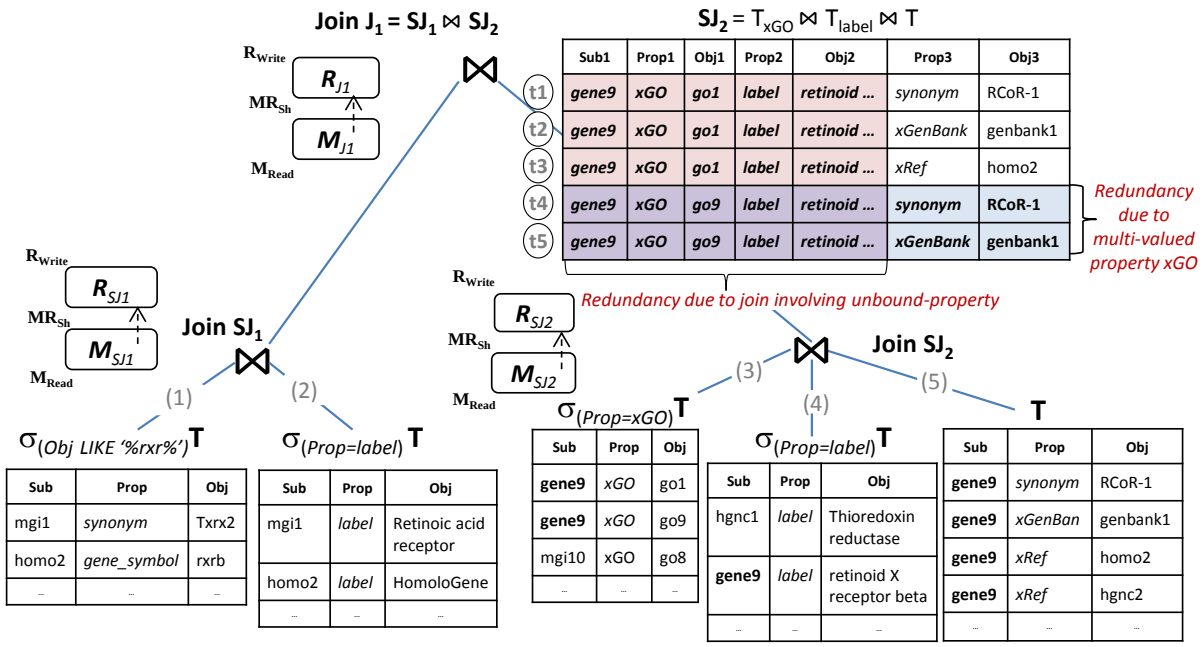


Figure 4.1: A MapReduce workflow for an unbound-property graph pattern query $Q1$ with two star subqueries SJ_1 and SJ_2; Join result of unbound-property star subpattern SJ_2 contains redundant information related to bound properties (xGO, label)

Query Q1

```
SELECT ?s1, ?label1, ?o1, ?label2, ?o2
WHERE {
 ?s1   ?p1   ?o1 .      (1)
FILTER regex(?o1,"rxr")
 ?s1   label ?label1 .  (2)
 ?s2   ?p2   ?s1 .      (3)
 ?s2   xGO   ?o2 .      (4)
 ?s2   label ?label2 .  (5)
}
```

Comments

Retrieve gene ontology(GO) terms related to "rxr', a gene of interest in analyzing Parkinson's disease. Q1 contains two star subpatterns, SJ_1 (1-2) and SJ_2 (3-5). (1) matches triples whose object contains "rxr" (any property). (3) specifies an unknown relationship connecting the two star subpatterns in the query.

Other than querying scenarios in integrated data warehouses, subqueries with unbound-property triple patterns are also generated by query rewriting techniques, that optimize ontological queries by rewriting them as a union of conjunctive queries. Examples of unbound-property queries can be found in real [67] and synthetic Semantic Web benchmarks [22], as well as other studies [64, 90]. In fact, 84% of queries in [4] are unbound-property queries.

Given a triple relation T and subset relations T_{xGO} and T_{label} with property types xGO and $label$, respectively, the subquery SJ_2 can be evaluated using relational joins ($T_{xGO} \bowtie T_{label} \bowtie T$). Figure 4.1 (right) shows the subrelations of T participating in $SJ2$ and a snapshot of the star-

join result. An issue with intermediate results in such cases is redundancy. For example, the result for SJ_2 in Figure 4.1(top right) contains repeated occurrences for matches of the bound properties – xGO and $label$, with each match of the unbound-property triple pattern. Further, the numbers of matches for the unbound-property triple pattern could be large if properties in the input dataset have high multiplicity ($gene9$ is associated with multiple $xRef$), further aggravating the issue of redundancy. High-multiplicity properties are common in real-world social networks as well as biological datasets such as Uniprot and Bio2RDF, e.g., some Uniprot properties have multiplicity as high as 13K.

For applications with periodic scale–up requirements, the growing trend is to employ cloud-processing platforms, e.g., Hadoop [20], Dryad [48], Hive [94], Pig [70], that are based on the MapReduce [37] computing model. However, any redundancy in intermediate results impacts query processing costs, particularly for MapReduce based distributed processing platforms that involve shipping of intermediate results across the network. The intermediate result footprint also impacts additional costs associated with sorting phases, materialization between the 2-steps of a MapReduce (MR) execution cycle, and total disk space requirements to store all intermediate states for fault-tolerance purposes. Hence, it is critical to minimize the footprint of intermediate results.

4.1.1 Related Work

Optimizing Relational Query Plans on MapReduce: There have been several efforts to shorten the length of MR workflows [8,10,46,72,102] to minimize the overall costs of MapReduce-based processing, share scans [68, 69, 99] and computations [38, 68] across MR workflows, cost-based and transformation-based MR workflow optimizer [60], and data skew problems [54]. Multi-way join algorithms [10,102] cluster multiple joins into a single [10] or few [102] MR cycles, but have not been applied to join-intensive workloads. Amongst the MapReduce-based RDF processing systems, SHARD [82] uses initial MR cycles to cluster triples into star subgraphs, followed by separate MR cycles to process each clause in the SPARQL query. HadoopRDF [46] pre-processes triples using the vertical-partitioning (VP) [6] approach, and uses heuristics to greedily group non-conflicting joins in a query to minimize the required number of MR cycles. However, unbound-property queries would require processing a union of all VP relations. The HadoopDB-based extension [45] uses a hybrid database-Hadoop architecture that exploits the partitioning scheme to push part of the execution into the database/RDF-3X. Hash partitioning on Subject can enable local evaluation of unbound-property star subpatterns. However, once the execution is handed over to Hadoop the redundancy in intermediate results impacts the disk I/O, sorting, and communication costs for the rest of the execution workflow. In order to minimize the data shuffle costs, MRShare [68] enables sharing of map output data across group-

ing operations on a common input relation. Some other works proposed a value-partitioning scheme [63] to manage reducer-unfriendly groups during the cube computation process, and a reducer-routing strategy [97] that groups map keys to balance the data across reducers. The lazy and partial β-unnesting strategies proposed in this work are in similar spirit.

Optimizing unbound-property queries: Earlier studies [84,87] have shown that vertical-partitioning(VP) [6] storage model may be inefficient for unbound-property queries. Such queries result in multiple joins and large unions of VP relations, which gets worse for data containing large number of property types. The multi-indexing schemes in systems such as RDF-3x [64] could benefit single-star unbound-property queries. However, such systems may not scale well for large RDF graphs, for queries with low selectivity and unbound objects [46]. There have been efforts [91] to optimize simple unbound-property queries to RDF views over relational databases. Since naive translation of an unbound-property query into SQL results in unions of multiple subqueries, the proposed Group Common Term transformer [91] exploits common terms in complex disjunctive SQL queries and rewrites them into a smaller number of queries. Our work proposes a scalable solution for processing unbound-property queries on MapReduce-based parallel processing platforms.

4.1.2 Contributions

In the previous chapter, we showed how NTGA's nested data model can be used to represent intermediate results more concisely, when compared to the normalized representation of intermediate results of relational operations. Specifically, n-tuples resulting from a relational star-join involving a multi-valued property are implicitly represented as a single triplegroup in NTGA, e.g.,

{ (*gene*9, *xGO*, **{go1, go9}**) // A single triplegroup representing
(*gene*9, *label*, retinoid...) // two n-tuples $t1$ and $t4$
(*gene*9, *synonym*, RCoR-1)} // by nesting object component

Though the "nested object" model and *nesting-aware physical operators* introduced in the previous chapter, reduce the I/O footprint of execution workflows, a join involving an unbound-property triple pattern would still produce 'n' triplegroups (assuming n triples with subject *gene*9). More importantly, all n triplegroups contain redundant bound-property component. In this chapter, we generalize the concept of triplegroup nesting to allow *nesting of property-object components*, to implicitly represent intermediate results while evaluating unbound-property queries. However, such an implicit representation involves triples playing multiple roles, i.e., a triple may match the bound and the unbound component of a query, which needs to be incorporated into the "unnest" process, referred here after as β-`unnest`. Additionally, there are implications of when and what portion of a triplegroup is β-unnested during the different phases

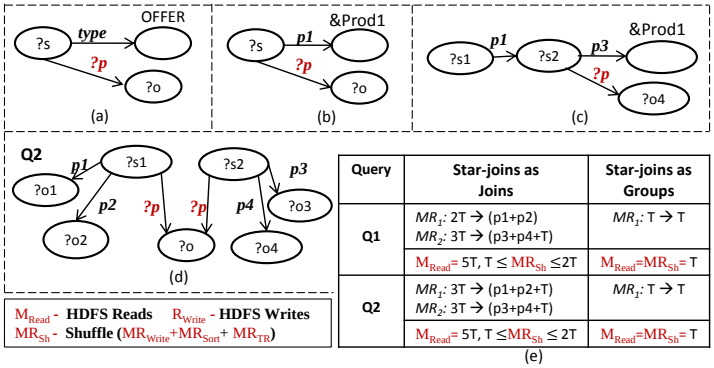


Figure 4.2: (a-d) Example unbound-property graph pattern structures (e) Analysis of HDFS reads, writes, and shuffle costs during the star-join computation phase

of an execution workflow, which are considered as part of choices for evaluation strategies. This chapter makes the following specific contributions:

- We introduce new NTGA logical operators and query rewrite rules that allow the translation of unbound-property queries into NTGA-based logical plans. The correctness and sufficiency of query rewrite rules is also presented.
- We introduce new NTGA physical operators that offer different evaluation strategies - *eager vs. lazy β-unnesting* of intermediate results during query processing.
- Extensive evaluation using large RDF graphs from both Semantic Web synthetic benchmark and real-world biological datasets demonstrates the efficiency of our approach over the relational-style processing of unbound-property graph patterns in default Pig and Hive systems.

4.2 Relational-style processing of Unbound-property Graph Pattern Queries on MapReduce

This section discusses the specific case of graph pattern queries involving unbound properties and their impact on the size of intermediate results and costs of MapReduce based processing.

4.2.1 Evaluating Unbound-property Queries using Relational Joins

When a triple pattern involves an unbound property (a variable in the Property position), it is called an *unbound-property triple pattern*. A graph (star) pattern with at least one unbound-property triple pattern is called an *unbound-property graph (star) pattern*. Unbound properties can be used to denote 'don't care edges' or unknown relationships in a graph pattern query. For

example, an unbound-property triple pattern can be used to describe all attributes of a Product resource ($\&Prod1, ?p, ?o$) or all entities related to a particular Vendor ($?s, ?p, \&Vendor1$). Figure 4.2 represents some example query structures for unbound-property graph patterns. Basically, an unbound-property graph pattern can be partitioned into two components, (i) bound-property triple patterns that provide the mandate for the required fixed edges, and (ii) unbound-property triple patterns that denote the unknown or 'don't care edges' in the query.

Typically RDF data is stored as ternary relations T or can be vertically partitioned [6] into smaller relations T_{P_i} that contain triples in T that correspond to the property P_i. The VP storage model enables fast retrieval of triples with specific bound properties. The unbound-property star-join corresponding to $SJ2$ in our example query $Q1$ can be expressed using relational joins over relations ($T_{label} \bowtie T_{xGO} \bowtie T$), where the join with the unbound-property triple pattern is simply expressed as a join over all triples in T corresponding to all property types. In general, the join result of a graph pattern query containing 'k' bound properties $\{P_1, P_2, ..., P_k\}$ and a single unbound property is a tuple with 3(k+1) arity as shown in Figure 4.1.

We shall use the notation GQ=$\{P_1, P_2..., P_l\}$ to denote a basic graph pattern query with (l-1) join operations between the set of relations $T_{P_1}, T_{P_2}, ..., T_{P_l}$. Graph pattern queries could further involve `FILTER` clause for specifying filtering operations.

Processing Relational-style Joins on MapReduce

Given a basic graph pattern query $GQ = \{P_1, P_2..., P_l\}$, the corresponding MapReduce execution workflow comprises of a sequence of MR cycles $MR_1, MR_2, .., MR_n$ such that $1 \leq n \leq (l-1)$. The ordering of the MR cycles is based on the join dependency and the intermediate join relations are written into the HDFS and re-processed in the successive MR cycle. Our example graph pattern query in Figure 4.1 (a) can be computed in a total of 3 MR cycles: 1 each for the two star subpatterns Stp_1 and Stp_2 (MR_1 and MR_2 respectively), followed by another cycle (MR_3) to join the results of Stp_1 and Stp_2. Each MR cycle in the MR workflow involves costs associated with initial input data reads in the map phase (M_{Read}), the data shuffling costs between the mappers and reducers that involve local disk writes at the mappers (M_{Write}), sort-merge costs (MR_{Sort}) as well as network transfer costs (MR_{TR}), and finally the cost of writing the reduce output to the HDFS (R_{Write}). Additionally, the output of each MR cycle is saved onto the HDFS till the completion of the entire workflow. Tasks that are likely to produce large intermediate results tend to impose a high demand on the required disk space. In the next section, we provide an estimation model to capture the amount of redundant content that could arise during the computation of an unbound-property star-pattern, and discuss its implication on the MapReduce costs.

4.2.2 Redundancy in Intermediate Results of Unbound-property Queries and its Impact on MapReduce Costs

Note that unbound-property triple patterns with a bound object can be evaluated as a simple filter and are not likely to cause redundancy in intermediate results. Hence, for this discussion we focus on patterns with unbound objects.

Let $Stp_u = \{P_{bound}, P_{unbound}\}$ be an unbound-property star subpattern such that $P_{bound} = \{P_1, P_2, ..., P_l\}$ represents the set of bound properties, and $P_{unbound}$ be a single unbound property. Then Stp_u can be evaluated as a set of relational joins $(T_{P_1} \bowtie T_{P_2} \bowtie ... \bowtie T_{P_l}) \bowtie T$ on the Subject column. Let the star-join corresponding to Stp_u be assigned to the kth MR cycle (MR_k) in the execution workflow, and let Out_k denote its output. Then the resultant join tuples in Out_k contain some redundant information related to the bound properties $P_1, P_2, ..., P_l$, as was highlighted in the result of Stp_2 in Figure 4.1. If the average degree of an entity (Subject) is m, then the final join with the unbound-property triple pattern produces an average of m tuples, for each tuple in the joining relation. Therefore, the output size of MR_k can be estimated as:

$$size(Out_k) = size(T_{P_1 \bowtie P_2 \bowtie .... \bowtie P_l}) * m$$

The redundant data RD_k written to the disk during MR_k can be estimated as:

$$RD_k = size(T_{P_1 \bowtie P_2 \bowtie .... \bowtie P_l}) * (m-1) + RD_l$$

where $RD_l = \begin{cases} 0, & \text{if } \forall P_i \in \{P_1, ..., P_l\}, P_i \text{ is single-valued} \\ > 0, & \text{if } \exists P_i \in \{P_1, ..., P_l\}, P_i \text{ is multi-valued} \end{cases}$

denotes the redundant content in $(T_{P_1 \bowtie P_2 \bowtie .... \bowtie P_l})$. Thus, RD_k represents the size of the redundant columns that arise due to the presence of unbound or multi-valued properties in the current join operation. To capture the degree of redundancy in the intermediate results, we use the notion of redundancy factor $RedF_k$ for cycle MR_k defined as:

$$RedF_k = \frac{RD_k}{size(Out_k)}$$

Thus, $RedF_k$ represents the proportion of the redundant data that is written into the HDFS at the end of cycle MR_k.

The redundancy factor directly impacts the HDFS writes (R_{Write}) for MR_k. For workflows with multiple MR cycles, the output of the intermediate cycles are stored in the HDFS till the completion of all cycles in the workflow. Hence, $RedF$ affects the HDFS requirement of an execution workflow and may have significant impact on query performance. Additionally, $RedF_k$ also negatively impacts the scan costs (M_{Read}) and the shuffle costs (MR_{Sh}) of the subsequent MR cycle MR_m that processes the output O_k. Depending on the selectivity (Sel_m) of the operations processed in MR_m, its redundancy factor $RedF_m$ is a function of ($RedF_k$ * Sel_m). Hence, the redundancy factor has a ripple effect on the costs of reads, writes, sorting and the data transfer costs across a workflow with multiple MR cycles.

In this work, we build on the advantages of the TripleGroup data model and algebra for efficient evaluation of unbound-property graph pattern queries on MapReduce. Specifically, the semantics of the group-filter operator (σ^{γ}) requires all properties in the query structure to be bound. However, to capture more complex patterns, the algebra and the set of rewrite rules need to be extended. The following section introduces a number of extensions which allow us to relax the above constraint to provide an extended group filter semantics for evaluating unbound-property queries.

4.3 Rewriting Unbound-Property Queries using NTGA

Consider an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$ such that $P_{bnd} = \{P_1, P_2, ..., P_k\}$ represents the set of bound properties and P_{unbnd} represents an unbound property. Let T_{St_u} be the star-join result of relation $T(Sub, Prop, Obj)$ with vertically partitioned subset relations T_{P_1}, T_{P_2},...,T_{P_k}, and let $T_{St_u(s)}$ represent a subset of T_{St_u} with subject $Sub = s$.

$$T_{St_u(s)} = \sigma_{Sub=s}(T_{P_1} \bowtie T_{P_2} \bowtie ... \bowtie T_{P_k} \bowtie T)$$

The tuples in $T_{St_u(s)}$ have arity $3(k+1)$, where each property in St_u is associated with 3 columns in $T_{St_u(s)}$. Figure 4.3 represents the tuples in $T_{St_1(S1)}$ for a star-pattern St_1 with 2 bound properties P_1, P_2 and an unbound property. To determine how St_u will be evaluated using NTGA, it will be useful to develop some correspondence between $T_{St_u(s)}$ and a subject triplegroup in NTGA. Note that a single triple may play multiple roles (occur multiple times) in the result of an unbound-property star pattern – one as a match for the bound property and the other as a match for the unbound property. For example, $(s1, p1, o1)$ in Figure 4.3 occurs once for the join with T_{P_1} and once for the join with T. In the NTGA data model, such multiple occurrences are implicitly represented once. Therefore, this must be accounted for in the transformation process.

Continuing with the transformation process, let π_{P_i} and π_{P_u} denote the projection of the (Sub, Prop, Obj) columns corresponding to parent relation T_{P_i} with bound property P_i, and unbound property P_{unbnd} respectively. Let tg_s represent the set union of triples formed by the 3 columns, i.e.

$$tg_s = \pi_{P_1}(T_{St_u(s)}) \cup ... \cup \pi_{P_k}(T_{St_u(s)}) \cup \pi_{P_u}(T_{St_u(s)})$$

Figure 4.3 denotes the triples in tg_{S1} for our example star-pattern St_1, formed by the set union of the partitions π_{P_1}, π_{P_2}, and π_{P_u}. tg_s has the following properties:

(i) $\forall t_i, t_j \in tg_s$, the triples t_i, t_j agree on the subject column s.

(ii) $\exists$ non-empty subset of triples $tg_{P_i} \subseteq tg_s$ such that $tg_{P_i} = \pi_{P_i}(T_{St_u(s)})$, for each bound property $P_i \in P_{bnd}$.

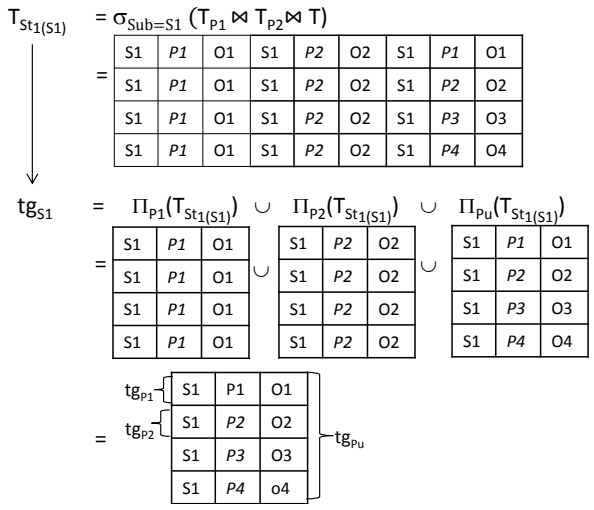


Figure 4.3: Transformation: n-tuples to a triplegroup

(iii) ∃ a non-empty subset of triples $tg_{P_u} \subseteq tg_s$ such that $tg_{P_u} = \pi_{P_u}(T_{St_u(s)})$ and $tg_{P_u} \cap (tg_{P_1} \cup ... \cup tg_{P_k})$ may be non-empty.

Essentially, the tuples in T_{St_u} can be horizontally partitioned into sets of tuples with the same Subject column, and each element in the partition can be vertically partitioned into 'triples' whose union is equivalent to a subject triplegroup tg_s in the NTGA data model. The use of set union instead of bag union ensures that we have a triplegroup. Further, subsets of triples in tg_s represent matches to the bound and unbound-property triple patterns in St_u. This process basically describes a sequence of translation steps from the relational algebra to NTGA. In other words,

$$\begin{aligned} T_{St_u(s)} &= \sigma_{Sub=s}(T_{P_1}) \bowtie ... \bowtie \sigma_{Sub=s}(T_{P_k}) \bowtie \sigma_{Sub=s}(T_{P_u}) \\ &= tg_{P_1} \bowtie ... \bowtie tg_{P_k} \bowtie tg_{P_u} \end{aligned}$$

Conversely, for our example star-pattern St_1 in Figure 4.3, tuples in $T_{St_1(S1)}$ are implicitly represented in tg_{S1} and can be produced by $(tg_{P_1} \bowtie tg_{P_2} \bowtie tg_{P_u})$. A useful property is to distribute the join with the unbound property triple pattern across a union of subset relations of T. In other words, if the triple relation T can be partitioned into two subset relations i.e. $T = \{T_{P'_u} \cup T_{P''_u}\}$. Then by the distributivity of join over union, we have:

$$T_{P_{bnd}} \bowtie (T_{P'_u} \cup T_{P''_u}) \equiv (T_{P_{bnd}} \bowtie T_{P'_u}) \cup (T_{P_{bnd}} \bowtie T_{P''_u})$$

Evaluating St_u using NTGA requires applying group filter (σ^γ) to match the required query structures. Recall that σ^γ is defined in terms of a set of bound properties. One might consider evaluating an unbound-property star-pattern query using σ^γ with a disjunction of concrete

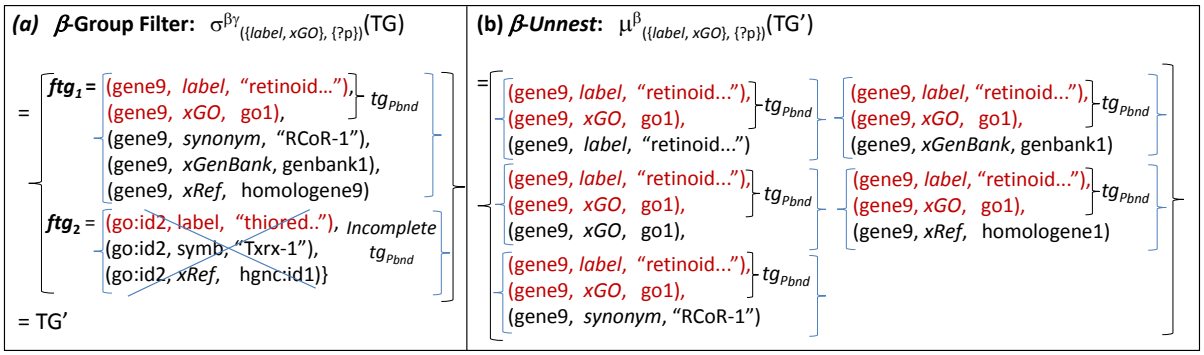


Figure 4.4: NTGA logical operators to evaluate unbound-property star-patterns

pattern combinations. Each such combination will consist of the set of bound properties P_{bnd} with each property in the database. For example, if $P_{bnd} = \{P_1, P_2\}$ is the set of bound-properties in the star pattern and $P = \{P_1, P_2, ..., P_{10}\}$ represents the set of all properties in the database. Then, the σ^{γ} expression is:

$$\sigma^{\gamma}_{(\{P_1,P_2,P_1\} \vee \{P_1,P_2,P_2\} \vee ... \vee \{P_1,P_2,P_{10}\})}(TG)$$

This would filter out triplegroups that do not match any of the required pattern combinations. However, the approach of enumerating all possible pattern combinations may be inefficient depending on the number of properties in the database. Additionally, the subject triplegroup tg_s may contain additional triples relevant to other patterns, and hence may not exactly match a single pattern combination. Hence, there is a need to relax the σ^{γ} to restrict the matching of structural constraints to the bound properties of the unbound-property star pattern. This means that triplegroups that contain all the bound properties (may contain additional properties), should be produced as part of the result for σ^{γ}. Once this is done, we need to extract subsets of triples in tg_s that are exact matches for any of the required pattern combinations. This is achieved by extracting the subset of triples corresponding to P_{bnd} and generating their union with each triple in the unbound-property subset tg_{P_u}. In the following section, we provide the formal definitions for a specialized group-filter operator ($\sigma^{\beta\gamma}$) and the unnest operator (β-`unnest`) that extracts the perfect matches to the unbound-property star-pattern. From here on, we assume the convenience function $tg.props()$ ($st.props()$) to retrieve the set of properties in the triplegroup tg (star pattern st).

Definition 4.3.1 *(β **Group Filter**) Given a set of subject triplegroups TG and an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$, the β **group-filter** operator $\sigma^{\beta\gamma}$ returns the subset of triplegroups in TG that contain a non-empty subset of triples matching all bound properties P_{bnd}. Specifically,*

$$\sigma^{\beta\gamma}_{(P_{bnd},P_{unbnd})}(TG) := \{\ tg_i \in TG \mid P_{bnd} \subseteq tg_i.props()\}$$

Essentially, $\sigma^{\beta\gamma}$ ensures that triplegroups contain a matching triple for each of the bound properties in P_{bnd}. Additionally, triplegroups may also contain triples containing other property types. For example, given $P_{bnd} = \{label, xGO\}$, triplegroup ftg_1 forms a valid result for the $\sigma^{\beta\gamma}$ expression in Figure 5.5(a). However, ftg_2 does not contain a matching triple for the bound property xGO and hence gets filtered out.

Definition 4.3.2 *(β **unnest**) Given a set of triplegroups TG and an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$, the **unnest** operator μ^{β} creates a set of triplegroups that are exact matches to St_u. Specifically,*

$$\mu^{\beta}_{(P_{bnd},P_{unbnd})}(TG) := \{\ tg_i = \{tg_{P_{bnd}} \cup t_i\} \mid tg_{P_{bnd}}, t_i \subseteq tg,\ tg_{P_{bnd}}.props() = P_{bnd},\ tg \in TG\ \}$$

In other words, the β-unnest operator extracts subsets of triples in a triplegroup tg that match the different pattern combinations corresponding to the unbound-property star-pattern. Figure 5.5(b) shows the 5 perfect triplegroups that are produced by β-unnesting the triplegroup ftg in Figure 5.5(a), each containing a subset of triples $tg_{P_{bnd}}$ matching the set of bound properties P_{bnd}, and a triple t_i that matches the unbound-property triple pattern.

Lemma 1 *Given a triple relation T and an unbound-property star pattern $St_u = \{P_{bnd}, P_{unbnd}\}$ such that the set of bound properties $P_{bnd} = \{P_1, P_2, ..., P_k\}$ and P_{unbnd} represents a single unbound property, the following equivalence holds:*

$$(T_{P_1} \bowtie ... \bowtie T_{P_k} \bowtie T) \cong \mu^{\beta}_{P_{bnd}}(\ \sigma^{\beta\gamma}_{(P_{bnd},P_{unbnd})}(\ \gamma_s(T)))$$

Proof: Let T_{St_u} and TG_{St_u} represent the set of tuples and triplegroups produced by evaluating an unbound-property star-pattern St_u using relational joins and NTGA respectively. We need to prove that all tuples in T_{St_u} are produced using NTGA. We prove by contradiction. Let us assume that there exists a tuple $tup_s \in T_{St_u}$ with subject s that cannot be produced using triples in tg_s. This can happen only if $\exists$ a triple $t_i \in tup_s$ such that $t_i \notin tg_s$. Firstly, since $t_i \in tup_s$, we know that the subject of t_i is s. If t_i.`props`() $\in P_{bnd}$, since t_i's subject is s, the $\sigma^{\beta\gamma}$ ensures that $t_i \in tg_s$. If t_i.`props`() $\notin P_{bnd}$, then $\sigma^{\beta\gamma}$ still retains t_i since its subject is s. Hence, $t_i \in tg_s$. The only other case is when a triple t_i plays multiples roles (matches both bound and unbound parts) which are implicitly represented in our data model. We rely on the correctness of the μ^{β} operator (illustrated earlier but proof omitted for brevity) to complete the proof.

Generalization to Multiple Unbound Properties. The β-unnest operator can be generalized to star-patterns containing multiple unbound-property triple patterns. Let P_{α}, P_{β},...,P_m represent the m unbound properties in a star-pattern. Then the β-unnest operator results in a set of triplegroups $\{tg_{\alpha\beta...m}\}$ each containing the bound-property subset $tg_{P_{bnd}}$ and m triples, one each matching the unbound properties P_{α}, P_{β},...,P_m.

$$\mu^{\beta}_{(P_{bnd},\{P_\alpha,P_\beta,..,P_m\})}(TG) := \{\ \{tg_{P_{bnd}} \cup t_\alpha \cup t_\beta \cup ...\cup t_m\}\ \}$$

such that $tg_{P_{bnd}} \subseteq tg$ is the bound-property subset, i.e., $tg_{P_{bnd}}.props() = P_{bnd}$, and triples $t_\alpha, t_\beta, ...t_m \in tg \in TG$.

4.4 Translation to MapReduce Plans

The logical operators proposed in the previous section are integrated into RAPID+ [51]. The query compilation process in RAPID+ begins with plans of logical operators, which are compiled to plans of physical operators, which could either be a single function or a function pair corresponding to the map and reduce phases of the logical operator. The MR plan is an assignment of physical operators to MR cycles.

The MR plan for an unbound-property query, executes the β-group-filtering using the `TG_UnbGrpFilter` ($\sigma^{\beta\gamma}$) operator in the reduce of the `TG_GroupBy`. This is followed by the β-`unnest` (μ^{β}) operator that produces a set of perfect triplegroups. Thus, both `TG_UnbGrpFilter` and `unnest` can be executed in the reduce of `TG_GroupBy` in a single MR cycle (MR_1). We call this as *eager* β-unnesting of triplegroups, represented in Figure 4.5(a). The joins between the triplegroups matching the different subpatterns can be computed using NTGA's `TG_Join` operator in the subsequent MR cycles. At the end of MR_1 for this strategy, we have intermediate results (perfect triplegroups for the star pattern subqueries) that contain redundancy with respect to the bound-properties. This increases the cost of $MR_1.R_{Write}$ and HDFS read ($MR_i.M_{Read}$) and shuffle costs ($MR_i.MR_{Shuffle}$) for subsequent cycles MR_i that process the output of MR_1. Therefore, optimization strategies to minimize the redundancy in intermediate results of the star-join computation phase would be useful to generate cost-effective MapReduce workflows.

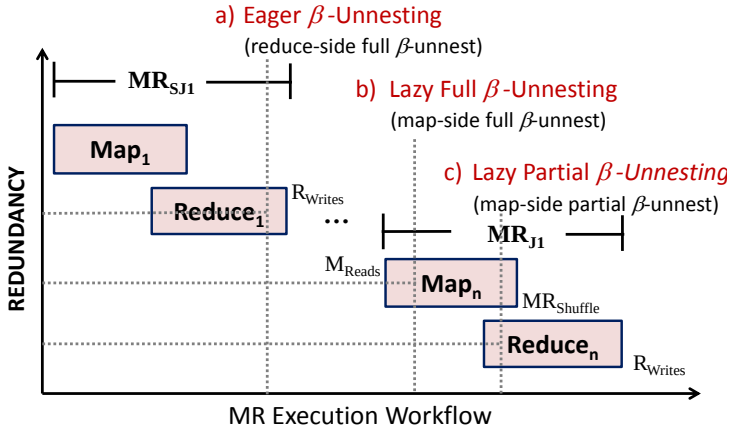


Figure 4.5: Choices for β-unnesting strategies, (a) eager β-unnest of a triplegroup during star-join, (b) lazy full β-unnest and (c) partial β-unnest in later join phase

4.4.1 Optimization using β-Unnesting Strategies

The intuition is to concisely represent the result of an unbound-property star-pattern as far along the MR workflow as possible. Unbound-property query structures such as `B4` in Figure 4.7 do not involve further joins based on the bindings of the unbound-property triple pattern, and thus can remain in its (nested) implicit representation till the end of the MR workflow. Query structures such as our example query `Q1` participate in joins based on the Object column of the unbound-property triple pattern. Hence, the star-join results for such star subpatterns need to be β-unnested before the join, since the map phase of `TG_Join` tags the triplegroups based on the join key and partitions them to the different reducers. We propose evaluation strategies to delay the β-unnesting of triplegroups.

Lazy Map-side β-Unnest: The β-unnesting of triplegroups can be delayed to a MR cycle that requires join on an unbound-property triple pattern, such as cycle MR_{J1} in Figure 4.5(b). Specifically, we push the β-`unnest` operator to the map phase of the corresponding `TG_Join` operator. We refer to the new physical operator as `TG_UnbJoin` (reduce phase remains same as `TG_Join`). By delaying the β-unnesting of triplegroups, we can minimize the redundancy in the results of the star-join computation phase, and hence avoid unnecessary writes, reads, and shuffle costs for all subsequent intermediate MR phases. However, the β-`unnest` operator expands the map output of `TG_UnbJoin`, which impacts the shuffling costs. Assuming that `TG_UnbJoin` is assigned to the kth MR cycle MR_k in the workflow, then the redundancy in map output impacts ($MR_k.M_{Write} + MR_i.MR_{Sort} + MR_i.MR_{TR}$).

Lazy Map-side Partial β-Unnest: We illustrate this strategy using Figure 4.6 . In order to support efficient look-up of (Property, Object) pairs in a triplegroup, we use an optimized internal representation scheme (extended multi-map) represented here as $AnnTG$, that concisely represents annotated triplegroups. Example annotated triplegroup $AnnTG_{gene9}^{\{o1,o2,o3,o4,o5\}}$ in Figure 4.6 represents the subject triplegroup ftg_1 (Figure 5.5(a)) which is a valid match for the unbound-property star subpattern SJ_2 in query $Q1$. Annotated TG $AnnTG_{gene9}^{\{o1,o2,o3,o4,o5\}}$ contains 2 bound-property triples (matching *label* and xGO) and 5 triples matching the unbound-property triple pattern. A β-unnest operation produces 5 triplegroups (all containing the same bound-property component) that form a part of the map output for MR_k. The default partitioning scheme in Hadoop assigns the map output tuples to a reducer r based on the hash value of the join key, i.e., $(hash(joinKey)\%r)$. In the case that we have just 2 reducers, it is possible that triplegroups containing redundant bound-property component are partitioned and assigned to the same reducer, based on the join keys (object of triples in the unbound-property component). For example, $AnnTG_{gene9}^{\{o1\}}$ and $AnnTG_{gene9}^{\{o3\}}$, may be assigned to the same Reducer, e.g., *Reducer1*. The redundancy in the map output of MR_k can be minimized if triplegroups that are eventually assigned to the same reducer, are concisely represented during the shuf-

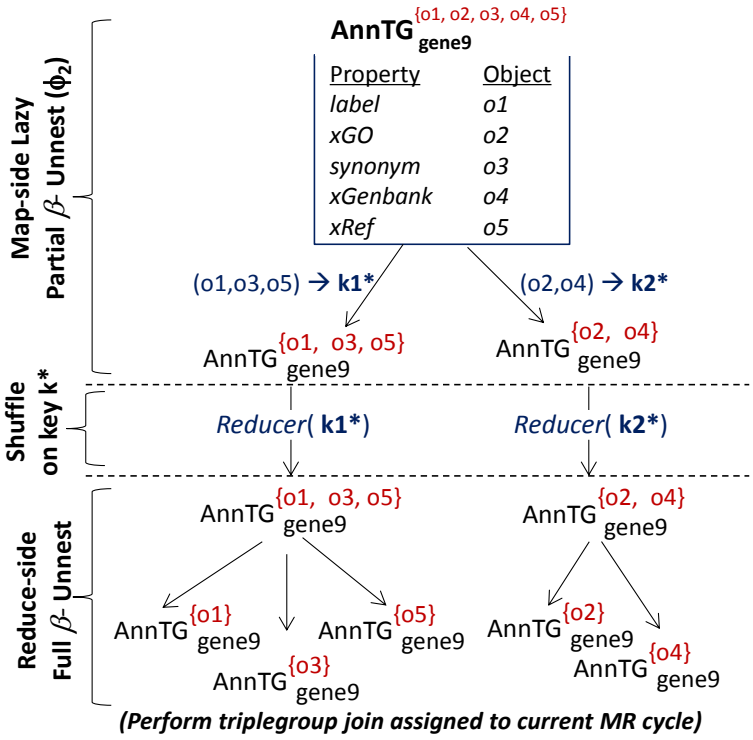


Figure 4.6: Lazy partial β-unnesting (ϕ_2)

fle phase, i.e., they are not β-unnested completely. By avoiding part of the β-unnesting, we can reduce the size of map output, and hence reduce the shuffling costs. We propose a partial β-unnesting strategy that creates a set of triplegroups that each contain the bound-property component ($tg_{P_{bnd}}$), and a subset of the unbound-property component $tg_{P_{unbnd}}$.

Definition 4.4.1 *(**partial** β**-unnest**) Given a set of triplegroups TG, an unbound-property star-pattern* St_u=$\{P_{bnd}, P_{unbnd}\}$*, and a partition function* ϕ_m *that partitions the triples in* $tg_{P_{unbnd}}$ *into m partitions, the **partial-**β**-unnest** operator* $\mu^{\beta'}$ *produces a set of triplegroups such that:*

$$\mu^{\beta'}_{(P_{bnd},\phi_m)}(TG) := \{\ tg^i = \{tg_{P_{bnd}} \cup partition_i\}\ \}$$

where

- $\forall\ tg \in TG$, the bound-property subset $tg_{P_{bnd}} \subseteq tg$ such that $tg_{P_{bnd}}.props() = P_{bnd}$.
- A function ϕ_m assigns a triple $t_j \in tg_{P_u} \subseteq tg$ to $partition_i$, i.e., $\phi_m : t_j \rightarrow partition_i$, where $i \in \{1, 2, ..., m\}$.

The function ϕ partitions the triples in $tg_{P_{unbnd}}$ into m buckets based on the value of the join key (object). Essentially, $\mu^{\beta'}$ produces a maximum of m triplegroups for each triplegroup $tg \in$

Algorithm 4: MR job workflow for NTGA plan

Job_1: Compute 'matching' triplegroup equivalence classes
  Map:
    `TG_GroupBy`.Map(Tuples T);
  Reduce:
    $TG \leftarrow$ `TG_GroupBy`.Reduce(Sub, List <Tuples>);
    $TG' \leftarrow$ `TG_UnbGrpFilter`(TG, $<EC, \{P_{bnd}, P_{unbnd}\}>$);
Job_i: Join between triplegroup equivalence classes
  Map:
    `TG_OptUnbJoin`.Map(TG') // partial β-unnest
    or `TG_UnbJoin`.Map(TG') // β-unnest
  Reduce:
    $TG'' \leftarrow$ `TG_OptUnbJoin`.Reduce(TG');
    or $TG'' \leftarrow$ `TG_UnbJoin`.Reduce(TG');

TG. For example, a partial β-unnest on $AnnTG_{gene9}^{\{o1,o2,o3,o4,o5\}}$ in Figure 4.6 using the partition function ϕ_2 produces 2 triplegroups - $AnnTG_{gene9}^{\{o1,o3,o5\}}$ and $AnnTG_{gene9}^{\{o2,o4\}}$ respectively. This implies that $\phi_2(o1) = \phi_2(o3) = \phi_2(o5) = k1^*$. Similarly, $AnnTG_{gene9}^{\{o2\}}$ and $AnnTG_{gene9}^{\{o4\}}$ are assigned to the same partition and hence remain implicitly represented as a single triplegroup. The redundant content in the map output is now a function of the partition range m. The partially β-unnested triplegroups are tagged and assigned to the reducers based on the partition key k^*. Triplegroup join with lazy partial β-unnest is implemented as a new physical operator, `TG_OptUnbJoin`. Figure 4.5(c) represents how I/O footprint (redundancy) can be reduced by partial and delayed β-unnesting at map phase of cycle MR_{J1}.

Algorithms For Physical Operators:

Algorithm 4 gives an overview of the job workflow for two key phases in the NTGA plan – Job_1, that computes 'matching' triplegroup equivalence classes that match all star subpatterns in the query, and Job_i, that computes the join between the triplegroup equivalence classes.

Job_1: ***Compute 'matching' TG equivalence classes.*** The input to this job is a set of 3-tuples (triples) in the RDF database, and the output is a set of annotated triplegroups $AnnTG$ that match the star subpatterns in the query. In the map phase, each tuple is tagged based on the Subject component. In the reduce phase, all tuples corresponding to the same Subject component Sub are processed in the same reduce(), producing subject triplegroups. This is followed by a group-filtering phase to filter out triplegroups that violate the structural constraints in the query. Algorithm 5 shows the pseudocode for the β group-filtering operator, `TG_UnbGrpFilter`. The (Property, Object) pairs in a triplegroup (`tempMap` in line 1), are matched with all equivalence classes (star subpatterns) in the query (line 2). For each matching equivalence class EC, the bound properties P_{bnd} are extracted (line 4). The tuples in the group are considered relevant to the query only if they contain the bound properties (lines 5-9). If the matched equivalence class contains an unbound-property, the resultant $AnnTG$ contains all

Algorithm 5: `TG_UnbGrpFilter`

```
    β-GrpFilter (tg, ECList:<EC,{P_bnd, P_unbnd}>);
1   tempMap ← extract triples in tg;
2   matchedECList ← match(tempMap, ECList);
3   foreach EC ∈ matchedECList do
4       P_bnd ← extract bound properties in EC;
5       if tempMap.keySet ⊆ P_bnd then
6           if EC contains unbound property then
                // β group filtering
7               propMap ← tempMap ;
            else
                // Extract only bound properties in EC
8               propMap ← extract P_bnd entries from tempMap;
9           emit ⟨AnnTG(Sub, EC, propMap)⟩;
```

the (Property, Object) pairs for subject Sub (lines 6-7). If the matched equivalence class does not contain any unbound-property, only the relevant (Property, Object) pairs that match the bound properties in the star subpattern are retrieved into the resultant triplegroup (line 8). Essentially, a group of tuples that does not contain the required set of bound properties for any of the star subpatterns in the query is filtered out.

Job_i**: *Join between TG equivalence classes.*** The input to this phase is a set of annotated triplegroups, belonging to the two equivalence classes whose join is to be computed. The output is a set of annotated triplegroups, representing the joined result between the two equivalence classes. Based on the amount of redundancy in intermediate results due to the unbound-property star subpattern, a decision is made to either enable a partial or full β-unnest of the map output. Star subpatterns where the unbound-property is associated with a (partially) bound object, are not likely to cause redundancy, and hence a full β-unnest is enabled (`TG_UnbJoin` operator). For all other cases, the `TG_OptUnbJoin` operator is used.

Algorithm 6 shows the map-reduce functions for the `TG_OptUnbJoin` operator that integrates lazy partial-β-unnest operation. In the map phase, the annotated triplegroups that join on Subject are tagged using the Subject's partition key k^* computed using ϕ_m (lines 1-3). For joins on Object, the AnnTG is partially β-unnested using the `partial-`β`-unnest` operation. The `partial-`β`-unnest` operator splits the (Property, Object) pairs in the triplegroup atg based on the Object's partition key resulting in a list of partially-unnested AnnTGs ($rMapList$ in line 4). A map output tuple is generated for each partially-unnested AnnTG, tagged by its partition key k^*(lines 5-7). The replication factor Rep is now a function of ϕ_m. In the reduce phase, all AnnTGs corresponding to the same group key k^* but different join keys are processed in the same reduce(). In order to selectively join them based on the original join key, the AnnTGs corresponding to the right relation ($rightEC$) are β-unnested into perfect triplegroups and hashed based on the join key ($rightHash$ in line 9). The algorithm iterates through each

Algorithm 6: `TG_OptUnbJoin`

```
   Map (key:null, val: AnnTG atg) ;
 1 if join on Sub then
 2     k* ← φ_m(atg.Sub);
 3     emit ⟨ k*, atg⟩ ;
   else if join on Obj then
 4     atgList ← partial-β-unnest (atg, φ_m) ;
 5     foreach partialMap ∈ atgList do
 6         k* ← extract k* for partialMap ;
 7         emit ⟨ k*, partialMap⟩ ;

   Reduce (key:k*, val:List of AnnTGs TG') ;
 8 leftList ← β-unnest leftEC AnnTGs from TG';
 9 rightHash ← β-unnest rightEC AnnTGs from TG';
10 foreach leftAnnTG ∈ leftList do
11     foreach prop ∈ leftAnnTG.propMap do
           // Handle multi-valued property
12         objList ← extract prop's objects from leftAnnTG ;
13         foreach joinKey ∈ objList do
14             rightAnnTG ← rightHash.get(joinKey) ;
15             emit ⟨ joinTGs(leftAnnTG, rightAnnTG)⟩;
```

AnnTG in the left relation ($leftEC$ in line 8), and probes the hashed relation ($rightHash$) based on the Object value (join key) for each property (lines 10-14). Multi-valued properties have multiple Object values and the probing is done for each value (lines 12-13). When a match is found, the two AnnTGs are joined (line 15) as per the definition of `TG_Join`. The partition factor used by ϕ depends on the size of input, potential redundancy factor, and average number of tuples that can be processed by a reducer.

4.5 Empirical Evaluation

We evaluate the proposed algebraic optimization technique on both real-world and synthetic datasets, and compare it with two popular relational-style MapReduce systems, Apache Pig and Hive. For NTGA, we evaluated two approaches for processing unbound-property graph pattern queries – *EagerUnnest* described in section 4.4, and the optimized *LazyUnnest* approach with map-side lazy β-unnesting. Experiments were conducted on NCSU's VCL [83], where each node in the cluster was a dual core Intel X86 machine with 2.33 GHz processor speed, 4G memory and running Red Hat Linux. 60 and 80-node Hadoop clusters (block size set to 256MB, 1GB heap-size for child jvms) were used with Pig release 0.11.1, Hive 0.10.0 and Hadoop 0.20.2. All results recorded were averaged over two or more trials.

Choice of Systems: Both Pig and Hive evaluate star-joins in a single MR cycle (one-star-join-per-cycle), resulting in same length workflows for all queries. Hive enables shared-scan of input relations within an MR cycle, thus minimizing the overall HDFS reads. Pig can

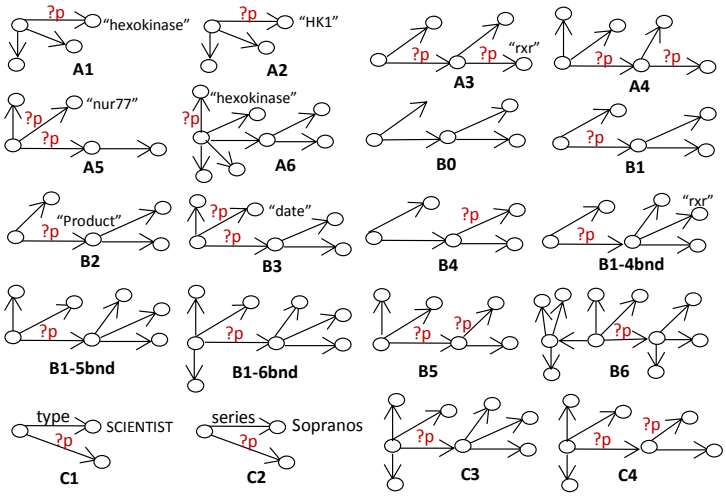


Figure 4.7: Testbed unbound-property RDF queries

execute independent MR cycles concurrently, which is beneficial while evaluating multiple star subpatterns. NTGA approaches produce shorter workflows (all-star-joins in single MR cycle) when compared to Hive/Pig for queries with multiple star subpatterns. A triple relation is loaded as a 3-column table in Hive, where as Pig (and NTGA) processes them as flat files. In Pig, the `SPLIT` operator is used to generate vertically-partitioning relations. HadoopRDF [46] does not currently support unbound-property queries and is not included for evaluation. Systems such as HadoopDB [45] scale well but rely on a heavy pre-processing phase that is more suitable for private clusters and less-evolving data. We focus on on-demand and pay-as-you-go workloads that involve quick exploration of datasets to get a sense of the data.

4.5.1 Setup

Environment: Experiments were conducted on NCSU's VCL [83], where each node in the cluster was a dual core Intel X86 machine with 2.33 GHz processor speed, 4G memory and running Red Hat Linux. 60 and 80-node Hadoop clusters (block size set to 256MB, 1GB heap-size for child jvms) were used with Pig release 0.11.1, Hive 0.10.0 and Hadoop 0.20.2. All results recorded were averaged over two or more trials.

Testbed - Dataset and Queries: Real-world life sciences data from Bio2RDF [18] was used for evaluation. The queried biological data warehouse integrated 24 datasets, consisting of a total of ∼4.7 billion triples (615GB in n-triple format). Two other real-world datasets, DBPedia Infobox (DbInfobox) [16] dataset of size 4.4GB (33.74M triples: 20.5M properties, 13.23M types) and the Billion Triple Challenge 2009 dataset (BTC-09) [3] of size 193GB (1.5B triples), were also used for evaluation. More than 45% of properties in both datasets are multi-valued with varying multiplicity. Two synthetic datasets generated by the BSBM [22] data

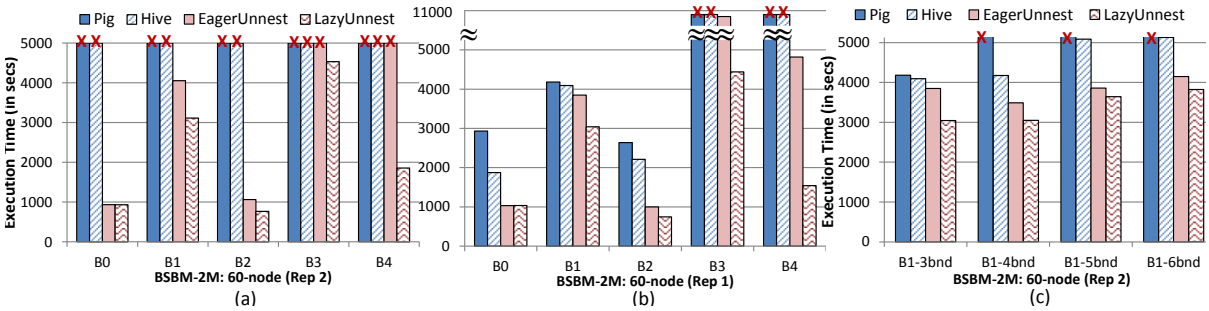


Figure 4.8: Performance with varying unbound-property star patterns with (a) replication factor 2 (b) replication factor 1 (c) Performance with varying size of bound-property component (BSBM-2M, 172GB, 60-node)

generator tool – BSBM-1M (85GB dataset with 1 million Products, total ∼370 million triples) and BSBM-2M (172GB dataset with 2 million Products, total ∼700 million triples) were used for scalability study. The evaluation tested unbound-property queries with varying selectivity, varying join structures (single join to more complex structures with multiple star subpatterns) that are represented in Figure 4.7. Graph patterns in queries `A1-A6` have been extracted from Bio2RDF demo queries [4]. Additional details about the evaluated queries, along with the Pig / Hive scripts, are available in Appendix C.

Varying join structures (`B1-B6`): Scalability experiments were conducted to evaluate different join structures with varying number of unbound-property triple patterns, and varying arity of star subgraphs. Figures 4.8(a) and (b) show a performance comparison of Pig, Hive, and the NTGA approaches for two-star queries with no unbound properties (`B0`), one unbound-property triple pattern with join on unbound object (`B1`), one unbound property associated with a partially-bound object (`B2`), two unbound-property triple patterns in the same star with one partially-bound object (`B3`), and an unbound-property triple pattern (`B4`). Pig / Hive evaluate all three queries using 3 MR jobs (one per star-join), while NTGA evaluates them in 2 MR jobs. The queries involve a multi-valued property *prodFeature* that impacts redundancy.

In order to avoid data loss during node failure, fault-tolerant systems such as Hadoop rely on replication of data blocks on multiple nodes using a configurable parameter (dfs.replication). Initial set of experiments were conducted using a replication factor of 2 for the larger dataset BSBM-2M on a 60-node cluster (1.6TB disk space). The results, shown in Figure 4.8(a), demonstrate how critical it is to concisely represent intermediate results and eliminate redundancy when possible. Missing bars marked with 'X' represent failed execution. Pig / Hive approaches failed during the last job (join between stars) for all 5 queries due to shortage of disk space. While *EagerUnnest* successfully executed for `B0`, `B1`, and `B2` by concisely representing subgraphs involving multi-valued properties, it failed for queries `B3` and `B4`. This is because the double unbound-property triple patterns in `B3` result in materialization of large intermediate results

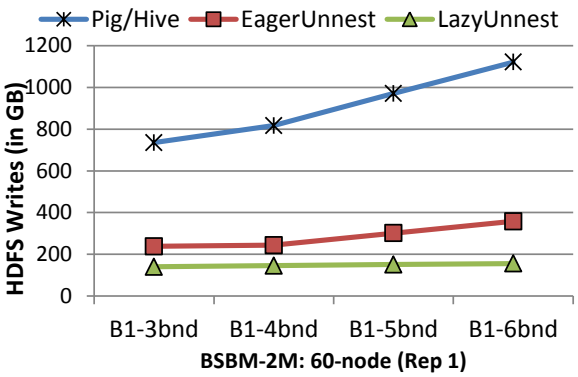


Figure 4.9: Total HDFS writes with varying size of bound component (BSBM-2M, 60-node)

during the star-join computation phase, and we see the benefit of pushing the β-unnesting to a later phase (*LazyUnnest*) in executing this query. Similarly, for query `B4`, *LazyUnnest* successfully executes by materializing concise intermediate results, while other approaches fail.

In order to analyze the performance of the different approaches on the larger dataset, the same set of queries were repeated after reducing the HDFS replication factor to 1. Figure 4.8(b) shows the results comparing the performance of the approaches for BSBM-2M on the same 60-node cluster. In general, we see the benefit of the NTGA approaches for all queries. Query `B0` shows a baseline case with all bound properties where Hive and NTGA approaches outperform Pig due to scan-sharing. Further, NTGA approaches concisely represent results containing multi-valued property which leads to I/O savings. For query `B1` (join on unbound-property triple pattern), lazy partial β-unnesting reduces the shuffle costs and is 21% faster than eager β-unnesting (27% faster than Pig and 26% faster than Hive). For query `B2`, all approaches evaluate the filter on the partially-bound object associated with the unbound-property triple pattern in the initial map phase, and from there on, the execution is similar to the baseline query `B0`. As in the case of replication factor 2, Hive and Pig failed again for `B3` and `B4`. The star subpattern with double unbound-property triple patterns (one with partially-bound object) in `B3`, is concisely represented in *LazyUnnest* with 80% less HDFS writes than *EagerUnnest*. In queries, such as `B4`, where the unbound-property triple pattern does not participate in join between stars, the lazy β-unnesting strategy keeps the result compact till the end, thus saving on intermediate disk reads / writes as well as final writes. Lazy β-unnesting using *LazyUnnest* results in 61% less HDFS writes than *EagerUnnest*, and overall has a 68% gain in performance times over the eager β-unnesting approach.

Choice of Lazy β-Unnesting Strategies: Testbed queries consist of varying structure of unbound-property triple patterns. For example, unbound-property triple patterns in queries $B2$ and $B3$ have partially-bound objects, i.e., the user does not know the exact property relationship

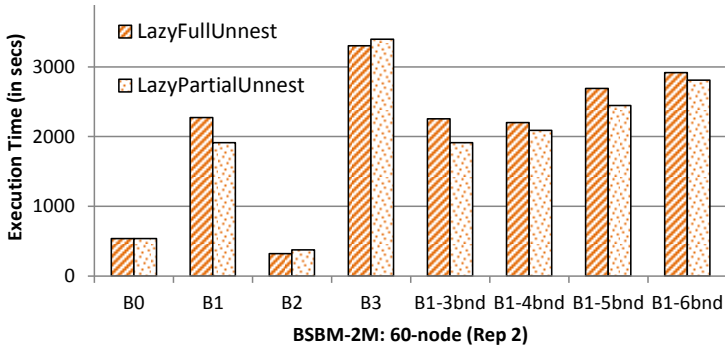


Figure 4.10: Lazy Full vs. Lazy Partial Unnesting: A comparative study of savings and overhead in MR cycle MR_{J1}

but knows something about the object. In such cases, it is likely that the number of triples matching the unbound-property triple pattern are reduced and hence the associated star-join is more selective, i.e., results in less number of pattern combinations when compared to same triple pattern with an unbound object. Other queries such as $B1$ consist of unbound-property triple pattern with an unbound object. Though lazy β-unnesting is beneficial for all cases, we wanted to study benefits and overhead of lazy full and lazy partial β-unnest strategies. Figure 4.10 shows execution times for the last MR cycle (MR_{J1}) where the join involving the unbound-property triple pattern is computed. Since the size of input MR_{J1} is same for both approaches, this analysis allows us to zoom into the map-side overhead for full and partial β-unnest, savings in shuffle costs, and analysis of reduce-side overhead in the case of partial-β-unnest. Our experiments show that a lazy full β-unnest may be sufficient for unbound-property queries with partially bound objects (queries $B2$ and $B3$). However, unbound-property queries with an unbound object ($B1$ series), benefit from partial-β-unnest. Other experiments were corroborative to these findings, and hence the *LazyUnnest* approach reported in this section evaluate lazy full-β-unnest for unbound-property queries with partially-bound-object patterns, and lazy partial-β-unnest for those with unbound-object patterns.

Varying number of bound-property edges: Unbound-property queries with bound-property triple patterns varying from 3 (`B1-3bnd`) to 6 (`B1-6bnd`) were evaluated. Figure 4.9 shows the total amount of HDFS writes for Pig, Hive and the NTGA approaches for the test queries evaluated on a 60-node cluster with BSBM-2M. In general, the increase in the number of bound-property components results in a gradual increase in the size of reduce output for Pig and Hive, while lazy β-unnesting keeps the result concise till the end of map phase of the last MR job (Job_2). The relational approaches produce 10 combinations of the bound component for the test queries since the relational arity of the subgraph that matches the unbound-property subpattern is 10. However, *LazyUnnest* compactly captures all the required

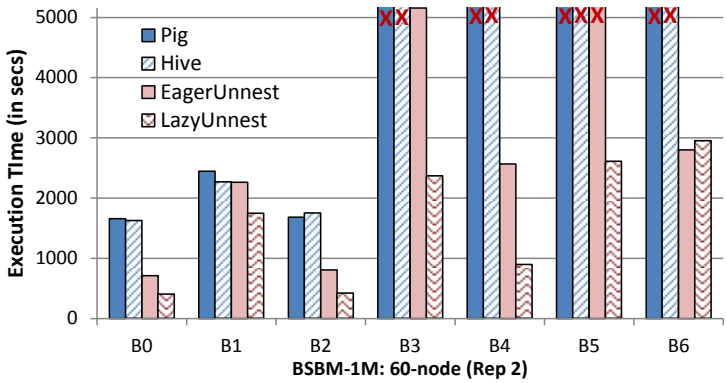


Figure 4.11: Performance comparison (BSBM-1M, 85GB)

combinations, resulting in approx. 80 to 86% less HDFS writes than Hive / Pig for queries `B1-3bnd` to `B1-6bnd`, respectively. Additionally, the reduce output for the NTGA approaches remain almost constant for such query patterns, which allows more flexible exploration of large datasets. Figure 4.8(c) shows a comparion of the execution times for all approaches. Note that Pig failed for all queries beyond three bound-property subpatterns. *LazyUnnest* (ϕ_{1K}) consistently outperformed the other approaches, running about 25% faster than Hive.

Varying size of RDF graphs: Figure 4.11 shows the evaluation of the BSBM queries using BSBM-1M (85GB) on the 60-node cluster (HDFS replication factor 2). NTGA approaches successfully executed for all datasets, with up to 80% less HDFS writes after the star-join computation phase for query `B1` when compared to Hive. Once again it was observed that both Pig and Hive failed for queries `B3` and `B4` due to insufficient disk space. This is due to the high redundancy in star-join result that ripples into the next MR job, impacting the scan and I/O costs. For query `B2`, *LazyUnnest* outperforms all other approaches, executing about 75% faster than both Pig and Hive. *LazyUnnest* reduces the redundancy in intermediate results, and thus improves the execution time of the eager β-unnesting approach (*EagerUnnest*) by 54% (65%) for query `B3` (`B4`). Hive / Pig failed to execute for more complex queries such as `B5` and `B6`. For `B6`, *LazyUnnest* has gain over *EagerUnnest* for first join between stars (unbound property), overall comparable. These sets of experiments demonstrate benefit of the proposed strategies in mitigating the effect of redundancy on MapReduce processing costs.

Real-world Unbound-property Queries (`A1-A6`): Figure 4.12 shows a performance comparison of Pig, Hive, and the two NTGA approaches for Bio2RDF queries `A1-A6` on a 80-node Hadoop cluster. Queries `A1` and `A2` have one star subpattern with one unbound-property triple pattern associated with partially-bound objects. For query `A1`, while Hive / Pig approaches produce all combinations of subtuples matching the bound property with triples matching the unbound property (∼63K tuples), *EagerUnnest* produces ∼7K triplegroups that concisely rep-

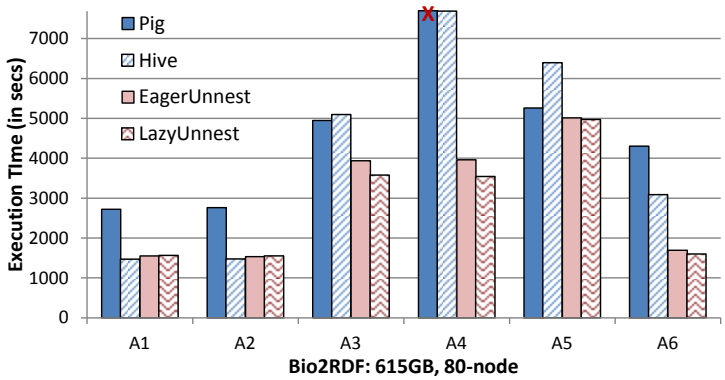


Figure 4.12: Evaluation of real unbound-property queries (Bio2RDF Life Sciences Dataset)

resent subtuples with multi-valued properties. *LazyUnnest* achieves more concise representation of all combinations corresponding to the unbound-property star pattern and produces only ∼3K triplegroups. The impact of the savings in HDFS writes due to elimination of redundancy in intermediate results, becomes more clear with the two-star queries (`A3-A6`).

Queries `A3` and `A4` contain an unbound-property in each of the two star subpatterns (one with partially-bound object). While Pig / Hive materialize 26GB of intermediate results in the star-join computation phase for query `A3`, the NTGA approaches write only about 1.3GB of data to the HDFS, contributing to the 32% performance gain over Hive while computing the star subpatterns. *LazyUnnest* results in reduced HDFS writes in Job_1, and reduced scan costs and shuffling costs in Job_2, resulting in additional 18% performance gain over *EagerUnnest* in Job_2. For query `A4`, Pig initiates 4 MR jobs (initial map-only job to read entire input and compress it, 2nd and 3rd MR jobs to compute the two star patterns, and the 4th job to join the stars). However, Pig approach failed (marked as 'X') due to lack of HDFS space while executing the last job. Again, there is a huge savings in terms of HDFS writes, with *EagerUnnest* and *LazyUnnest* producing only 1.8GB and 0.6GB of intermediate results, respectively, after the initial star-join phase, as opposed to 152GB of writes in Hive. An important factor that results in large intermediate results with relational-style processing, is the redundancy due to the presence of large number of high multiplicity properties in biological datasets (representative of real-world datasets). For `A4`, *EagerUnnest* and *LazyUnnest* approaches are 48% and 53% faster than Hive, respectively.

Query `A5` contains a star pattern with two unbound-property triple patterns – one whose object matches a gene "nurr77", and the other with an unbound object, connecting the star to a single edge retrieving the *label* property type. Hive executes `A5` using 2 MR jobs, with both jobs requiring a full-table scan. NTGA approaches also execute using 2 MR jobs but with one full-table scan, resulting in overall savings of about 1400s (22% gain) over Hive. The single

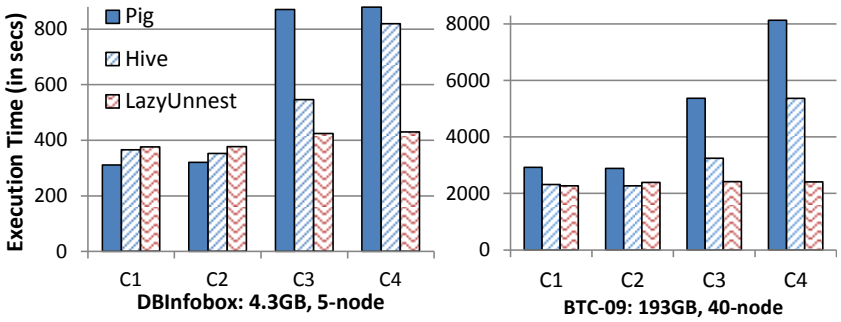


Figure 4.13: Evaluation of real-world unbound-property queries (DBInfobox and Billion Triple Challenge'09)

unbound-property triple pattern in query `A6` partially binds the object to "hexokinase". While Hive uses 3 MR jobs, including 2 for the star-join computation, Pig uses an extra map-only job to compress the input (total 4 jobs). NTGA's *LazyUnnest* approach shows a benefit of up to 48% over Hive.

DBPedia Queries (`C1`-`C4`): Additional experiments were conducted on varying sizes of real-world datasets, 4.3GB DBInfobox dataset (5-node cluster) and 193GB BTC-09 dataset (40-node cluster) as shown in Figure 4.13. Four different query structures were used. `C1` and `C2` are simple queries with single join that retrieve all information about Scientists (unselective), and Sopranos TV series (selective). In the case of DBInfobox dataset, since the data processed is quite small, the benefit of the NTGA approach is not seen for the first two queries. However, Pig does better than Hive since it processes two copies of the input relation, and hence initiates double the number of mappers and reducers. `C3` and `C4` represent real-world scenarios during exploration where the relationship between entities (star subpatterns) is unknown. NTGA approaches showed a performance gain of 20-22% and 50% over Hive and Pig respectively for query `C3`, and resulted in approx. 80% less HDFS writes than Hive. All four queries had redundancy factor greater than 0.6. In particular, `C4` which involved an unbound-property in each of the two star patterns showed a redundancy factor close to 0.89, and hence showed major improvement (50% gain over both Pig and Hive) with the lazy β-unnesting strategy.

Unbound-property queries on the BTC-09 dataset resulted in very large HDFS reads which negatively impacted Pig the most, due to its multiple scans per star-join. The scan-sharing across star patterns in NTGA resulted in 50% less HDFS reads for the two star queries. NTGA approaches resulted in 54% (25%) gains over Pig (Hive) for query `C3` with 1 unbound-property. The result of the star-join phase for `C4` (2 unbound properties) has redundancy factor of 0.93(0.75GB) and increases to 0.98(14GB) in the final output for Pig/Hive. The lazy β-unnesting strategy results in 98% less HDFS writes, and have 70% (55%) performance gain over Pig (Hive) for `C4`. In general, real-world data contained multiple multi-valued properties with varying multi-

plicity, and highly benefited by the generalized nested representation of triplegroups and lazy β-unnesting strategies while processing unbound-property queries.

4.6 Chapter Summary

In this chapter, we present a scalable solution for processing unbound-property graph pattern queries on MapReduce, by minimizing the redundancy in intermediate results that adds avoidable costs while processing long execution workflows. The proposed strategies concisely 'nest' the intermediate results and lazily 'unnest' them only when necessary. Experiments show promising results for different query join structures on real-world as well as synthetic benchmark datasets.

Chapter 5

Optimizing Complex Grouping and Aggregation Constraints in RDF Analytical Queries

5.1 Motivation

Growing amount of linked open data is enabling interesting applications that combine data from different domains for analysis. This leads to the need for analytical queries on large Semantic Web data warehouses and the recent extension to SPARQL [43] to include grouping-aggregation constructs is a recognition of that need. For example, the ReDD-Observatory [104] discusses a study reporting the total number of deaths and the number of clinical trials for Tuberculosis and HIV/AIDS in all countries, to analyze the disparity between biomedical research and the disease burden in developing countries. This study involved information about clinical trials and effectiveness of treatment options from *ClinicalTrials.gov*, statistics about mortality for different countries from the *Global Health Observatory* (GHO), published by the World Health Organization and biomedical research (MEDLINE publications and other life science journals) available in the *PubMed*. The results need to be grouped based on both country and disease, followed by aggregations on the number of clinical trials and deaths due to the concerned disease in each country. Another example is a Semantic Web application called AlzPharm [55], that queries several semantically-linked neuroscience datasets to find information relevant to studying neurodegenerative diseases, e.g., identify the different groups of drugs used for Alzheimer's Disease when grouped by their molecular targets and clinical usage.

Non-trivial analytical queries require multiple aggregations over different groupings of data, and often, some of the groupings are related or overlapping, resulting in redundant scans and joins over large relations. Consider the query shown in Figure 5.1(a), adopted from the Berlin

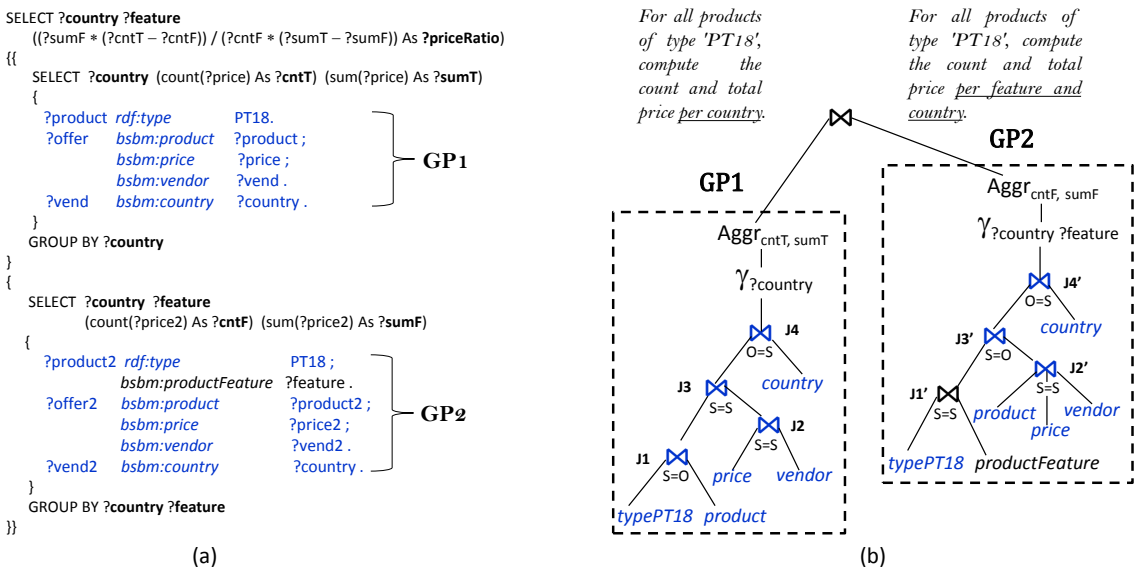


Figure 5.1: (a) An example SPARQL analytical query, Query 1: *For each country, retrieve product features with the highest ratio between price with and without that feature* (b) relational algebra based query plan for Query 1

SPARQL BI benchmark [2] to analyse the country-wise impact of product features on price of a product. The query involves two descriptions $GP1$ and $GP2$ for products of type '$PT18$', and two grouping constraints, one for each pattern, on *country* and on ($feature$, *country*) combinations, respectively.

Since RDF data is usually organized as binary or ternary relations, a graph pattern (usually with multiple join operations) is required to describe an n-ary relation over which a grouping is required. When a query involves multiple groupings, multiple such graph patterns will be present as subqueries. Further, if the groupings are related or overlapping, then there can be significant amount of overlap in the graph pattern subqueries. Figure 5.1(b) shows a summarized query plan with two major subqueries using the traditional evaluation technique: one subquery for *GP1* with four joins that matches subgraphs about product offers for products of type *PT18* ($J1$), the vendors offering products ($J2$-$J3$), their countries ($J4$) and prices of offers. This is followed by a grouping based on vendor's *country*. The second subquery contains a similar graph pattern *GP2* with five joins (due to the addition of product features in the pattern description) followed by a grouping on *feature-country*. Answers from the two subqueries are then joined to compute the final price ratio, resulting in a total of *10 join operations, and 2 grouping operations*.

In contrast, in the relational model, such OLAP queries are usually assumed to be evaluated over suitably organized schemas, such as star or snowflake schemas, consisting of n-ary relations. In this context, different optimization strategies ranging from specialized query constructs [25,

26, 28, 29, 41], efficient indexing [71, 101], materialized views [30, 42], and efficient evaluation in distributed data warehouses [12, 13] have been proposed. However, the absence of such schema organizations for RDF creates the need to investigate appropriate optimization strategies. One naive approach is to completely decompose the evaluation into two distinct phases, a graph pattern evaluation phase that constructs and materializes a suitable set of n-ary relations, followed by relational-style optimizations. However, such an approach prevents the possibility of optimizations across the two phases, e.g., early projections, partial aggregations, etc. Therefore, a holistic optimization strategy is likely to be more advantageous.

A promising direction is based on the observation made in [26] that relational expressions tightly couple grouping and aggregation specifications, often resulting in complex algebraic expressions that confound query optimizers. To address this issue, authors propose an alternative way of expressing such queries based on an operator, *MD-Join*, that decouples grouping and aggregation. In this chapter, we propose an approach that has a similar spirit by generalizing the *MD-Join* operator to deal with redundancy in overlapping graph patterns resulting from multiple grouping-aggregation specifications in RDF analytical queries. Further, with a focus to support large scale RDF analytics, this chapter overviews how execution of the query expressions formulated with such a generalization can be evaluated on MapReduce based Cloud platforms. We integrate the work on algebraic optimization of graph pattern queries described in the previous chapters, with algebraic optimization of OLAP queries to achieve a holistic optimization of RDF analytical queries. Next, we review some of the existing approaches that are relevant to the problem of optimizing RDF graph analytical queries.

5.1.1 Related Work

In order to enable better expression of online analytical processing (OLAP) queries, constructs such as the `CUBE BY` [41], grouping sets [25], unpivot [40], and MD-Join [26] have been proposed. We discuss some of them in more detail in Section 5.2.1 in the enabling more optimized evaluation of complex OLAP queries. In this section, we overview some other works that are relevant to distributed processing and RDF analytics.

Parallel and Distributed Evaluation of Relational OLAP Queries. Earlier work on parallel evaluation of aggregates proposed adaptive algorithms [86] to handle a range of grouping selectivities (ratio of result size to the input size) across queries. Subsequent research [12] on distributed evaluation of OLAP queries identified optimization strategies that exploit knowledge about data distributions to reduce the amount of data transfer between the local sites and the centralized coordinator of a distributed data warehouse. In the context of MapReduce, an approach that allows an overlapping redistribution scheme [33] was proposed to enable parallel evaluation of correlated aggregations with sliding windows. The MR-Cube [63] algorithm was

proposed for distributed cube computation of partially algebraic measures on MapReduce. A value-partitioning scheme was proposed to deal with reducer-unfriendly groups, i.e., cube groups that tend to increase the load on a reducer. The work on MR-Cube was integrated into Apache Pig 0.11 to support multi-level aggregations using the CUBE and ROLLUP functions. Similar efforts to support enhanced aggregations in Apache Hive, led to the inclusion of the GROUPING SETS clause (equivalent to multiple GROUP BYs connected by a UNION) as well as the CUBE / ROLLUP clauses. Such operations assume the existence of a fact table on which CUBE and other operations can be applied, which does not hold for RDF triples data.

Expression and Evaluation of RDF Analytical Queries. The RDF Data Cube vocabulary (QB) [36] was provided as a recommendation to enable publication of statistical data in RDF adhering to Linked Data principles. The work on Open Cubes vocabulary [39] enables representation of multi-dimensional data using RDF Schema (RDFS). Other extensions such as [49, 50] propose a multi-dimensional model based on QB to support OLAP queries and maps OLAP operations to SPARQL. Recent work on RDF analytics [35] proposed a way to define an analytical schema on RDF graphs to capture interesting information, and formalize analytical queries *AnQ* over such an analytical schema. The proposed *AnQ* separates the grouping and aggregation definitions, similar to the relational *MD-Join* operator. An earlier work proposed RAPID [89], that extends Pig's query primitives to support Map-Reduce based execution of the *MD-Join* operator. In the next section, we discuss in detail the benefits and issues in adopting the *MD-Join* operator for RDF graph analytics on MapReduce.

5.1.2 Contributions

In the previous chapters, we have considered the case of graph pattern processing using NTGA. While the relational approach often requires multiple scans of the same relation, the NTGA-based approach eliminates redundant scans during graph pattern processing phase, and can be beneficial while evaluating the multiple graph patterns required by an analytical query. Previous chapters did not consider grouping and aggregation clauses. In this chapter, we build on the foundations of sharing that is already inherent in the NTGA approach and enhance its benefits by enabling *MD-Join* based optimizations. Specifically, we make the following contributions:

- An algebraic rewriting of overlapping graph patterns (in a SPARQL analytical query) using a *composite graph pattern* based on common substructures. A decoupled reformulation of the grouping-aggregation definitions in an RDF analytical query expressed using a composite graph pattern.
- A set of logical and physical operators for efficient evaluation of a composite graph pattern, as well as parallel evaluation of independent aggregations on a composite graph

pattern. The suite of operators and optimizations are integrated into *RAPIDAnalytics*, an extension of Apache Pig.

- A comprehensive evaluation of RAPIDAnalytics using basic and multi-aggregation RDF analytical queries on a real-world as well as synthetic benchmark datasets.

5.2 Background and Challenges

5.2.1 Optimization of Complex OLAP Queries

Commonly, OLAP queries involve aggregating the measure attributes in a fact table, that is grouped by a set of dimension attributes. For example, a fact table containing *Sales* information, may be grouped by combinations of *country-feature* dimensions, and aggregated based on the *price* measure attribute. Decision-support systems require more complex OLAP queries involving multi-phase groupings and aggregations, that are difficult to express and optimize using basic relational grouping and aggregation operators. There has been a body of work to enable better expression and evaluation [12, 25–29, 40, 41] of complex OLAP queries. While constructs such as the `CUBE BY` [41], grouping sets [25], etc., allow a user to have a finer control over the group specifications and aggregations, other works focus on efficient indexing [71, 101], materialized views [30, 42], and efficient evaluation of OLAP queries in distributed data warehouses [12, 13]. The authors of *MD-Join* [26] demonstrated that the decoupling of the grouping definition and aggregation computations not only allows more succinct expression of complex OLAP queries, but also enables better optimization opportunities. For example, the decoupled reformulation of complex OLAP queries using the *MD-Join* eliminates redundant scans and joins involving fact table for computing slightly different groupings and aggregations. In the context of MapReduce-based processing, an implication of such reductions in joins and scans of large fact tables is efficient execution plans with savings in I/O, network transfer, and other processing costs. In order to illustrate the benefits and issues in adopting the *MD-Join* for RDF graph analytics on MapReduce, we first review the specifics of the *MD-Join* operator.

MD-Join Operator. The basic idea is to construct a set of base values B related to the interested groupings, and associate subsets of the detail relation R based on a condition θ. The subsets are then aggregated based on the list l. Below is the formal definition of the *MD-Join* operator as provided in [26].

Definition 5.2.1 *Let $B(\boldsymbol{B})$ and $R(\boldsymbol{R})$ be relations, θ be a condition involving attributes in $\boldsymbol{B}$ and $\boldsymbol{R}$, and l be a list of aggregate functions $(f_1, f_2, ..., f_n)$ over attributes c_1, c_2,...,c_n in $\boldsymbol{R}$. Then the MD-Join, $MD(B, R, l, \theta)$, is a relation with schema $(\boldsymbol{B}, f_1\_R\_c_1, ..., f_n\_R\_c_n)$, whose instance is determined as follows. Each tuple $b \in B$ contributes to an output tuple $\boldsymbol{b}$, such that:*

- $\boldsymbol{b}[A] = b[A]$, *for every attribute* $A \in \boldsymbol{B}$.
- *for each tuple* $b \in B$, *let* $RNG(b, R, \theta) = \{r \in R \mid \theta(b, r)$ *is true*$\}$. *Then, the value of attribute* $f_i\_R\_c_i$ *of tuple* $\boldsymbol{b}$ *is given by* $\boldsymbol{b}[f_i\_R\_c_i] = f_i\{\{t[c_i] \mid t \in RNG(b, R, \theta)\}\}$, *where* $\{\{...\}\}$ *denotes multiset.*

B is called the base-values relation and R is called the detail relation.

Note that both B and R can be a result of other relational algebra expressions.

Let us consider an MD-Join based rewriting of our example query. Let relation T_{p_i}(s,o) be a vertically-partitioned relation containing subject (s) and object (o) components of triples with property p_i. Let relation $T_{typePT18}$(s) contain subject resources of product type "PT18". Then, our example query in Figure 5.1 can be expressed in relational algebra using MD-joins as:

$$MD_2(\ B_2 = MD_1(\ B,\ F_1,\ \{\texttt{SUM}(price),\ \texttt{COUNT}(price)\},\ \theta_1),\ F_2,\ \{\texttt{SUM}(price),\ \texttt{COUNT}(price)\},\ \theta_2\ )$$

where

B: $\pi_{(o \to country)}(\ T_{country}\ ) \times \pi_{(o \to feature)}(\ T_{productFeature}\ )$

F_1: $\pi_{(T_{country}.o \to country, T_{price}.o \to price)}\ (\ R_1\ )$

where $R_1 = (\ T_{typePT18} \bowtie T_{product} \bowtie T_{price} \bowtie T_{vendor} \bowtie T_{country}\ )$

F_2: $\pi_{(T_{productFeature}.o \to feature, T_{country}.o \to country, T_{price}.o \to price)}\ (\ R_2\ )$

where $R_2 = (\ T_{typePT18} \bowtie T_{productFeature} \bowtie T_{product} \bowtie T_{price} \bowtie T_{vendor} \bowtie T_{country}\ )$

θ_1: $F_1.country = B.country$

θ_2: $F_2.country = B_2.country$ and $F_2.feature = B_2.feature$

The inner *MD-Join* expression MD_1 corresponds to graph pattern $GP1$ in Figure 5.1. The base-values relation B contains the country-feature combinations, along with place holders for sum ($sumT$) and count (cnT) of price. The detail relation F_1 contains details about the vendor's country and offer prices for products of type "PT18". As the detail relation F_1 is scanned, the corresponding entries in B are updated based on the condition θ_1. The output of the first *MD-Join* is a relation with schema ($country$, $feature$, $sumT(price)$, $cntT(price)$), which acts as the base-values relation B_2 for the second *MD-Join* (corresponds to graph pattern $GP2$). Relation F_2 is the detail relation for the second *MD-Join* MD_2, which contains details about the offers of product type "PT18", along with its features, and vendor's country. B is extended with place holders for sum and count of price per feature, i.e., $sumF(price)$ and $cntF(price)$. The detail relation F_2 is scanned and the corresponding entries in B_2 are updated based on the condition θ_2. The base-values relation is then scanned once to compute the final ratios.

Discussion: The decoupled re-formulation of multi-phase grouping-aggregation queries using MD-Join enables a more succinct expression. However, one of the primary goals of the

MD-Join operator is to reduce the number of scans of the large fact relation, which is not applicable in the context of RDF graphs. Below, we list down some of the issues with adopting the *MD-Join* operator for RDF analytics on MapReduce.

- Unlike traditional OLAP systems, where the fact and dimension tables are available and suitably organized into star or snowflake schema, the fine-grained data model in RDF necessitates *additional join operations to reassemble the relevant fact and dimension information*. For example, the detail relation F_1 would require four join operations to retrieve details about offers of product type "PT18", and its vendor's country information.

- Evaluating such a query plan on MapReduce would require *multiple MR cycles (one per star-join) to evaluate the join expressions computing the base and fact information.* For example, the computation of base-values relation B, and the two detail relations F_1 and F_2 required by the MD-Join-based rewriting compiles into an MR plan with 9 MR cycles. Further, each *MD-Join* operator can be compiled into a map-reduce cycle [89], thus resulting in a total of 11 MR cycles. Such a *sequential execution workflow limits opportunities to share scans* of input triple / VP relations.

- One useful optimization rule [11] of *MD-Join* is that series of independent *MD-Joins* can be executed in parallel, if both the *MD-Joins* involve the same detail relation and the aggregations are not dependent on a previous phase. However, RDF analytical queries are likely to involve different join expressions for the detail relations. Our example involves two different detail relations F_1 and F_2, and such an optimization is not directly applicable.

Thus, in order to fully exploit the benefits of such a decoupled reformulation, we require additional optimizations to share scans and computations across the graph pattern matching as well as grouping-aggregation phases.

5.2.2 Sharing in RDF Query Processing on MapReduce: Opportunities and Techniques

It is common for complex OLAP queries to compare and analyze subtotals across multiple dimensions. Such scenarios result in subqueries that compute groupings over an overlapping subset of dimensions. For example, $GP2$ requires groupings on $country\text{-}feature$ combinations, while $GP1$ computes a roll-up to include ALL product features, thus grouping only on $country$. In the context of RDF graph analytical queries, such related groupings result in subqueries with common subexpressions, i.e., graph patterns with overlapping structures, which opens up opportunities to share scans and computations.

Figure 5.2 summarizes some of the sharing opportunities while evaluating RDF graph analytical queries involving basic graph pattern BGP_i and grouping-aggregation Gi_{Aggi}. Let Stp_{abc}

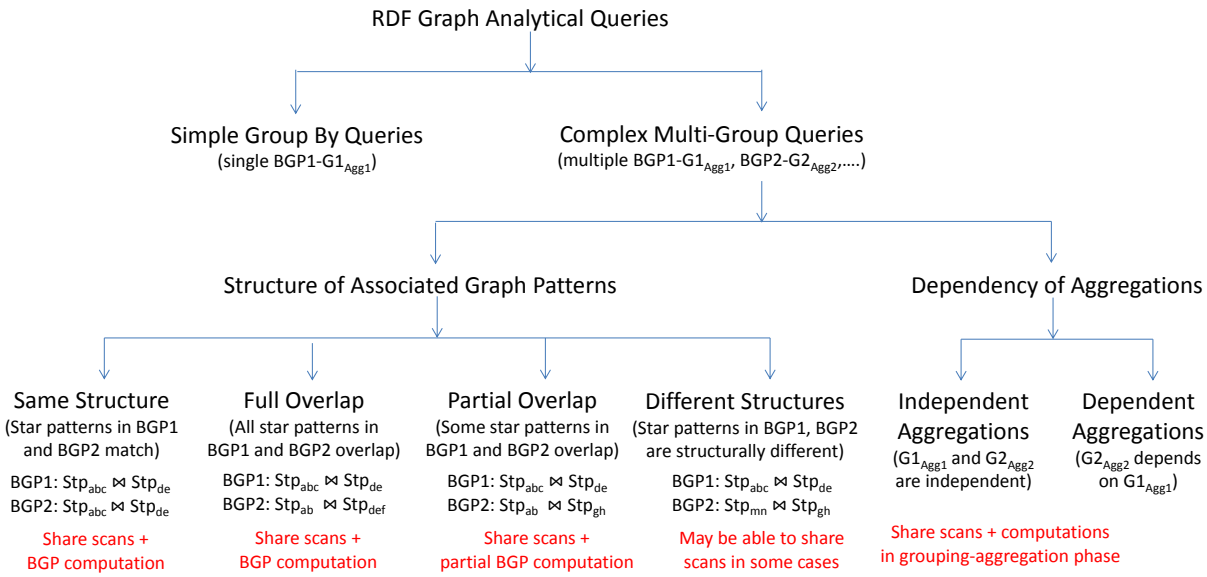


Figure 5.2: Sharing Opportunities while evaluating RDF graph analytical queries

denote a star pattern with properties a,b, and c. If the associated graph patterns in a query have the exact same structure (same join expression), the graph pattern can be evaluated only once, thus sharing scans and computations in MR cycles responsible for the graph pattern processing phase. In cases where graph patterns overlap (subsumption relationship in join expressions), there may be opportunities to rewrite the query in a way that allows sharing of scans and computations in MR cycles that evaluate common substructures. Additionally, there may be opportunities to share data references in intermediate results, thus reducing the costs of materializing intermediate results at mappers and reducers, as well as network transfer costs. Partially overlapping graph patterns may allow some shared-scans and computations, e.g., during evaluation of subpatterns Stp_{abc} and Stp_{ab}. In the case where the graph patterns are structurally different, there may be opportunities to share scans within a MR cycle, e.g, star patterns involving multiple joins of the same triple / VP relation. Further, sharing of scans is possible when the grouping-aggregations to be computed are independent and involve the same graph pattern.

Several optimization techniques have been proposed to reduce I/O and network transfer costs in MapReduce-based processing by sharing scans and computations [24, 52, 58, 68, 99]. MRShare [68] considers various techniques such as sharing scan of input data, sharing map functions, and sharing map output across multiple reduce functions, while executing multiple grouping queries on MapReduce. The work in YSmart [58] groups correlated operations in complex queries, e.g., `Joins` and `GROUP BYs` accessing the same table, into a single MapReduce job to reduce redundant scans, computations, and network transfers. The proposed correlation-

based optimizations are now integrated into Apache Hive 0.12.0.

Multi-query Optimization in SPARQL. A previous work [57] on multi-query optimization (MQO) of SPARQL queries, makes use of the SPARQL `OPTIONAL` clause. The `OPTIONAL` clause is used in SPARQL to allow querying of predicates that may not exist, i.e., answer is returned if there is a subgraph matching the `OPTIONAL` graph pattern, else it is ignored. Given an RDF graph and a set of graph pattern queries Q with common substructures, the basic idea of SPARQL MQO is to (i) rewrite the input queries into a set of queries Q_{OPT} with OPTIONAL clauses (represents non-overlapping structures), (ii) evaluate queries Q_{OPT} over the RDF graph, and (iii) distribute the results of Q_{OPT} to the input queries in Q. As a simple example, consider the star patterns rooted at subject variable ?*product* in $GP1$ and $GP2$ of Figure 5.1(a),

Q_1: `select * where { ?product rdf:type PT18 . }`
Q_2: `select * where { ?product rdf:type PT18; bsbm:productFeature ?feature .}`

Queries Q_1 and Q_2 can be rewritten using `OPTIONAL` clause as,

Q_{OPT}:
```
select * where { ?product rdf:type PT18 .
                 OPTIONAL { ?product bsbm:productFeature ?feature .} }
```

where the non-overlapping pattern in Q_2 is expressed using the `OPTIONAL` clause. Results matching original queries Q_1 and Q_2 are extracted from results of Q_{OPT}. Results in Q_{OPT} may have NULLs for products with no available feature information. Additionally, the result distribution process needs to taken into account that the presence of multi-valued properties in the optional component, may result in duplicate answers to the overlapping component.

Discussion. While processing complex analytical queries, a possible optimization strategy is to evaluate the individual graph patterns (without the grouping-aggregation clauses) using the SPARQL MQO approach, then extract the answers to the original graph patterns, and apply the groupings and aggregations on the corresponding query results. In the context of MapReduce-based processing, the MQO-based rewriting of graph pattern queries is likely to result in shorter MapReduce execution workflows, when compared to sequential execution of individual graph pattern queries. However, our experiments on Hive show that the MQO-based approach may not always be beneficial. Since Q_{OPT} is evaluated ahead of time, Hive is not able to apply optimizations such as early `PROJECT`s of columns based on the grouping-aggregation clauses. Additionally, if the query involves joins followed by grouping-aggregation, Hive enables partial aggregation of results in the *reduce*() of the final join, which reduces the amount of intermediate results that is materialized and passed to the next MR cycle. Such optimizations are also not applicable when Q_{OPT} is evaluated ahead of time. Note that Q_{OPT} would need to be evaluated and stored as a table, since Hive does not support logical views involving complex queries with multiple joins, nor does it support materialized views.

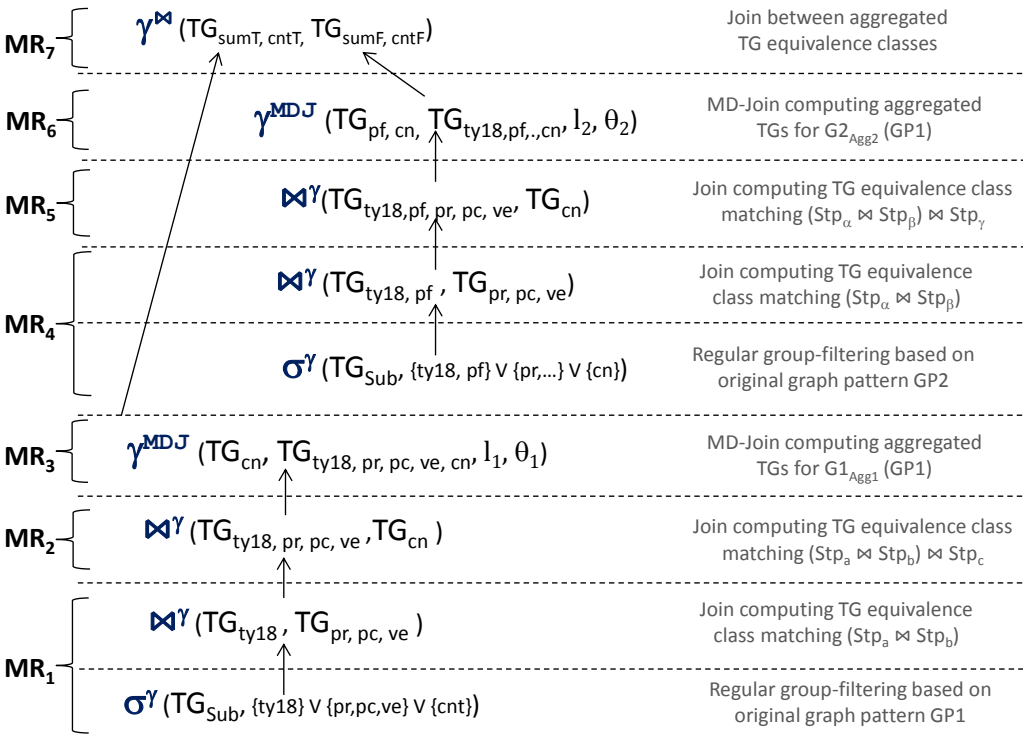


Figure 5.3: NTGA-based MR execution plan: Evaluating original graph patterns $GP1$-$GP2$ Prefixes: $ty18(typePT18)$, $pf(prodFeature)$, $pr(product)$, $pc(price)$, $ve(vendor)$, $cn(country)$

In order to overcome the limitations of the existing techniques, we propose a *holistic* approach that not only reduces the number of scans and join operations by enabling shared execution of the common substructures in an RDF analytical query, but also incorporates the benefits of the *MD-Join* approach. In order to achieve efficient execution plans for MapReduce based processing, we build on the advantages of the NTGA approach to achieve short execution workflows with reduced I/O footprint.

5.3 Algebraic Rewriting of RDF Analytical Queries

In this section, we first review the rationale to use NTGA-based execution plans for evaluating complex RDF graph analytical queries. We provide an intuition of our approach and list down the requirements for query rewriting and execution, to enable shared execution of RDF analytical queries involving multiple grouping-aggregation constraints.

5.3.1 Rationale

NTGA-based execution plans can be beneficial to shorten the MR execution plan while processing the individual graph patterns. Figure 5.3 shows the NTGA query plan and MR execution

plan for our example query using basic NTGA operators. For simplicity, we denote the basic graph patterns as $GP1$ and $GP2$, and the corresponding grouping-aggregations as $G1_{Agg1}$ and $G2_{Agg2}$, respectively. We denote the three subject star patterns in $GP1$ as Stp_a, Stp_b, and Stp_c, respectively. Similarly, the star subpatterns in $GP2$ are denoted as Stp_α, Stp_β, and Stp_γ, respectively. For the rest of this section, we assume a pre-processing phase (1 MR cycle) to group the input triples by subject Sub to compute the subject triplegroups TG_{sub}. The evaluation of our example RDF graph analytical query involves the following steps:

- *Computation of graph pattern* $GP1$ (MR_1-MR_2): Group-filtering of subject triplegroups TG_{Sub} to extract triplegroups matching the star patterns Stp_a (TG_{ty18}), Stp_b ($TG_{pr,pc,ve}$), and Stp_c (TG_{cn}), followed by a join ($\bowtie^\gamma$) between first two equivalence classes TG_{ty18} and $TG_{pr,pc,ve}$ in MR cycle MR_1. This is followed by another join between the result of MR_1 with equivalence class TG_{cn} in MR_2.

- *Computation of grouping-aggregation* $G1_{Aggr1}$ (MR_3): For this step, we assume an operator similar to *MD-Join* for TG equivalence classes (γ^{MDJ}) that would require a separate MR cycle MR_3 to compute the required aggregations on the TG equivalence class matching graph pattern $GP1$.

- *Computation of graph pattern* $GP2$ (MR_4-MR_5): Similar to $GP1$, this step also requires two MR cycles to compute the join between TG equivalence classes $TG_{ty18,pf}$ and $TG_{pr,pc,ve}$ in cyle MR_4, and joining the resultant TG equivalence class with TG_{cn} in cycle MR_5.

- *Computation of grouping-aggregation* $G2_{Aggr2}$ (MR_6): Grouping-aggregation computed using *MD-Join* (γ^{MDJ}) on TG equivalence class matching the graph pattern $GP2$.

- An additional MR cycle MR_7 is used to associate the grouping-aggregations computed on $GP1$ (output of MR_3) with those from $GP2$ (output of MR_6).

Note that NTGA-based execution of the example query results in an execution workflow with 7 MR cycles, as opposed to 11 MR cycles using the relational-style plan. However, further optimizations are possible by exploiting the common subexpressions in the associated graph patterns. Since the graph patterns $GP1$ and $GP2$ in example 1 have some overlapping structures, we can re-write the relational algebra expression using MD-Join as follows:

$$MD_2(\ B_2 = MD_1(\ B,\ F',\ \{\texttt{SUM}(price),\texttt{COUNT}(price)\},\ \theta_1),\ F',\ \{\texttt{SUM}(price),\texttt{COUNT}(price)),\ \theta_2\ )$$

where
B: $\pi_{(o \rightarrow country)}\ (\ T_{country}\ ) \times \pi_{(o \rightarrow feature)}\ (\ T_{productFeature}\ )$
F': $\pi_{(T_{productFeature}.o \rightarrow feature, T_{country}.o \rightarrow country, T_{price}.o \rightarrow price)}\ (\ R'\ )$

$R' = (\ T_{typePT18} \bowtie T_{product} \bowtie T_{price} \bowtie T_{vendor} \bowtie T_{country}\ ) \rtimes T_{productFeature}$

θ_1: $F'.country = B.country$

θ_2: $F'.country = B_2.country$ and $F'.feature = B_2.feature$ and $F'.feature$!= NULL

The join expressions in F_1 and F_2 can be rewritten as F' to capture common expressions in a way that eliminates redundant join operations. A left-outer join is used to allow missing values while joining with the non-overlapping expression, i.e., $T_{productFeature}$. Such a rewriting not only results in less numbers of join operations, mapping to fewer numbers of MR cycles, but also enables parallel evaluation of the two *MD-Joins* with a single scan of the common detail relation F'.

Our approach is based on query rewriting of RDF graph pattern queries with multiple grouping-aggregation queries by, (i) identifying an *overlap* between associated graph patterns based on structural constraints, (ii) evaluating a *composite graph pattern*, that can retrieve answers matching the original graph patterns in the query, thus sharing scans and computations during the graph pattern matching phases, and (iii) computing the required groupings and aggregations based on the composite graph pattern, in a way that allows opportunities to share scans and computations in the grouping-aggregation phase.

Identification of Overlapping Graph Patterns. In order to determine structural overlap between graph patterns, we need a way to compare the involved star pattern structures as well as the join structures that connect the star patterns. First, we consider the case of overlap between star patterns.

Definition 5.3.1 ***(Overlapping Star Patterns)*** *Let Stp_1 and Stp_2 be two subject-rooted, star graph patterns such that $props_1$ (resp. $props_2$) denotes the set of properties in the triple patterns in Stp_1. Further, let $L = props_1 \cap props_2$. If L is empty, then Stp_1 and Stp_2 do not overlap. Otherwise, they are considered overlapping subject to the following condition:*

- *For triple patterns, tp_1 = ($s1$, rdf:type, $o1$) in Stp_1 and tp_2 = ($s2$, rdf:type, $o2$) in Stp_2, if $o1 = o2$, i.e., the object components (types) are the same.*

Figure 5.4 represents our running example with graph patterns $GP1$ and $GP2$. Star pattern $Stp_a \in GP1$ overlaps with $Stp_\alpha \in GP2$ since both match on the object of *rdf:type* triple. Similarly, star patterns Stp_b and Stp_β overlap, and so do Stp_c and Stp_γ.

To generalize the notion of overlapping from star patterns to graph patterns, we need to capture similiarity of join structures between star patterns in terms of their joining triple patterns, i.e., the triple patterns that contain the join variables. For convenience we assume the existence of a function `var`(tp) that returns the set of variables in a triple pattern tp. Given two triple patterns tp_1 and tp_2, we say that a join variable $jv_1 \in$ `var`(tp_1) $\cap$ `var`(tp_2). A join variable

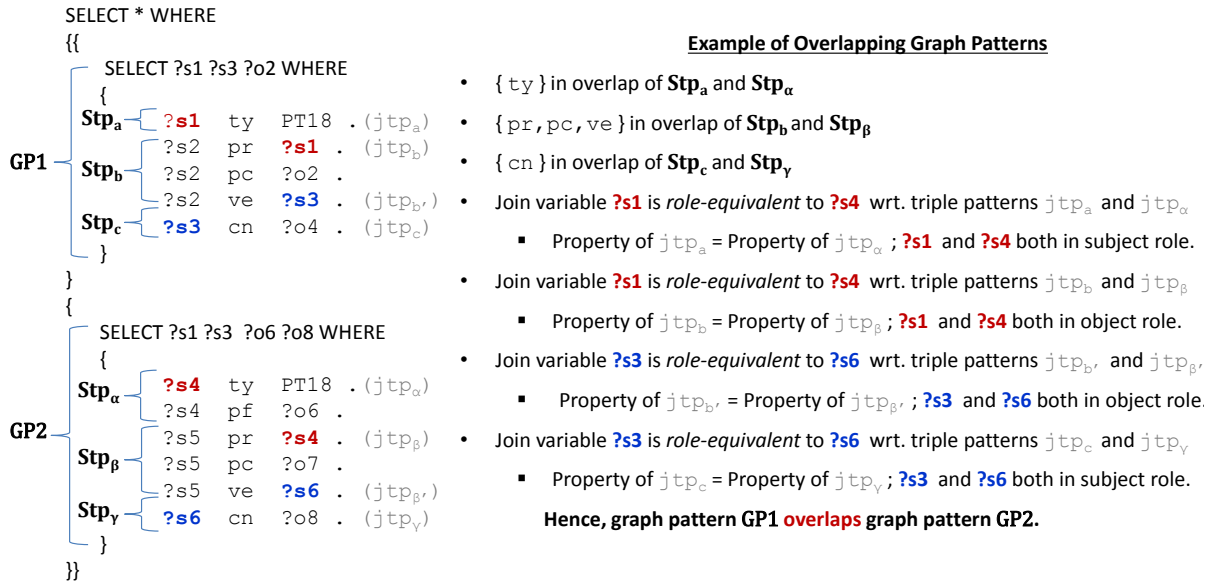


Figure 5.4: Examples of structural overlap in graph patterns

$jv_1 \in tp_1$ is *role-equivalent* to another join variable $jv_3 \in tp_3$ if property of tp_1 = property of tp_3 and jv_1 has the same role (i.e. is in same position in tp_1) as jv_3 is in tp_3.

Definition 5.3.2 ***(Overlapping Graph Patterns)*** *Let $GP1$ be a graph pattern with star patterns Stp_a and Stp_b such that the triple pattern $jtp_a \in Stp_a$ joins with the triple pattern $jtp_b \in Stp_b$ on join variable jv_a. Let $GP2$ be another graph pattern with star patterns Stp_α and Stp_β. Further, let Stp_a overlap with Stp_α, and let Stp_b overlap with Stp_β. Then, the graph patterns $GP1$ and $GP2$ are said to overlap if the following conditions hold:*

- Star patterns Stp_α and Stp_β join on triple patterns jtp_α and jtp_β such that join variable $jv_\alpha \in \texttt{var}(jtp_\alpha) \cap \texttt{var}(jtp_\beta)$.
- Join variables in the overlapping star patterns are role-equivalent, i.e., jv_a is role-equivalent to jv_α w.r.t. joining triple patterns jtp_a and jtp_α, as well as joining triple patterns jtp_b and jtp_β, respectively.

Graph patterns $GP1$ and $GP2$ in Figure 5.4 overlap since all three star patterns overlap and have the same join structures. The subject-object join structure between star patterns Stp_a and Stp_b in $GP1$ matches with the join structure connecting Stp_α and Stp_β in $GP2$. Similarly, join structures connecting Stp_b and Stp_c match those of overlapping star patterns Stp_β and Stp_γ.

Construction of a Composite Graph Pattern. The overlapping as well as the non-overlapping substructures in graph patterns $GP1$ and $GP2$ can be captured and expressed using a single *composite graph pattern* GP', which we define next.

Definition 5.3.3 ***(Composite Star Pattern).*** *Let $Stp_a \in GP1$ and $Stp_\alpha \in GP2$ be two overlapping star patterns consisting of triple patterns with the set of properties $props_a$ and $props_\alpha$, respectively, i.e., $props_a \cap props_\alpha$ is non-empty. Then, a composite star pattern Stp'_a is formed by combining the star patterns Stp_a and Stp_α, such that the set of properties $props'_a$ in Stp'_a is a union of the overlapping (primary) as well as the non-overlapping (secondary) properties in Stp_a and Stp_α, i.e.,*

$$props'_a = \{\ P_{prim} \cup P_{sec}\ \}$$

where

- $P_{prim} = \{\ props_a \cap props_\alpha\ \}$, is the set of *primary* properties that define the common substructures in Stp_a and Stp_α.
- $P_{sec} = \{\ p_i \mid p_i \in props_a \cup props_\alpha,\ p_i \notin P_{prim}\ \}$, is the set of *secondary* properties that define the non-overlapping substructures in Stp_a and Stp_α.

For the rest of this chapter, we denote secondary properties with an underline. Our example star patterns $Stp_a \in GP1$ and $Stp_\alpha \in GP2$ in Figure 5.4, can be re-written as a composite star pattern $Stp'_a = \{\ ty18, \underline{pf}\ \}$, where $ty18$ is the primary property and pf is the secondary property (underlined).

Overlapping graph patterns $GP1$ and $GP2$ can be re-written as a composite graph pattern GP' that consists of a composite star pattern for each pair of overlapping star patterns in $GP1$ and $GP2$. Our example graph patterns $GP1$ and $GP2$ in Figure 5.4 can be re-written as a composite graph pattern GP' such that:

$$GP' = \{\ Stp'_a,\ Stp'_b,\ Stp'_c\ \}$$

with composite star patterns $Stp'_a = \{\ ty18,\ \underline{pf}\ \}$, $Stp'_b = \{\ pr,\ pc,\ ve\}$ and $Stp'_c = \{\ cn\ \}$. Since the join structures (between stars) in overlapping graph patterns match, the same join structure holds for a composite graph pattern. Using the notion of a composite graph pattern, our example RDF analytical query with grouping-aggregations over $GP1$ and $GP2$ can b re-written as follows:

$$(GP1,\ G1_{Agg1}) \bowtie (GP2,\ G2_{Agg2}) \Rightarrow (GP',\ (G1_{Agg1},\ G2_{Agg2}))$$

such that the required grouping and aggregations can be computed based on the composite graph pattern.

Evaluation of a Composite Graph Pattern. Since a composite graph pattern subsumes the original graph patterns, answers matching both graph patterns $GP1$ and $GP2$ can be computed by evaluating the composite graph pattern GP'. However, the star-joins involving

secondary properties need special handling. Further, the presence of secondary properties in composite star patterns may result in superfluous subtuples when evaluating a composite graph pattern. Such subtuples do not match either of the original graph patterns, resulting in wrong aggregates. Hence, we need a way to encode valid join combinations based on the presence and absence of secondary properties. Based on these observations we list down the requirements for an NTGA-based rewriting of an RDF graph analytical query:

- In order to evaluate a composite star pattern on a set of triplegroups, we need a special group-filter operator that allows the presence of secondary properties that may or may not exist (optional).
- Given that we now have triplegroups matching a composite star pattern, we need an operator to extract subsets of triples in the triplegroup that are exact matches to the n original star patterns.
- While processing joins on triplegroups that match a composite star pattern, we need a set of conditions (let us refer to them as α) based on the secondary properties, e.g., pf is not null, to restrict the processing to relevant triplegroups that match either of the original star patterns.
- In order to evaluate groupings and aggregations based on a composite graph pattern, we need an *MD-Join* operator that processes sets of triplegroups, and computes the groupings and aggregations based on a given set of α conditions.

5.3.2 Logical Operators for Evaluation of RDF Graph Analytical Queries

Computing a Composite Star Pattern. The existing group filter (σ^{γ}) operator in NTGA matches the required query structure, defined by a set of properties. In order to support composite star patterns that include optional properties, we need to relax the σ^{γ} to restrict the matching of the structural constraints to the set of primary properties that are 'required'. Additionally, the new σ^{γ} operator must ensure that the resultant triplegroups contain no other properties, other than the set of 'required' properties and the set of 'optional' properties.

Definition 5.3.4 ***(Optional Group Filter)*** *Given a set of subject triplegroups TG and a star pattern $Stp_o = \{P_{prim}, P_{opt}\}$, the* ***optional group-filter*** *operator $\sigma^{\gamma_{opt}}$ returns the subset of triplegroups in TG that contain a non-empty subset of triples matching all properties in P_{prim}, and may contain triples matching properties in P_{opt}. Specifically,*

$$\sigma^{\gamma_{opt}}_{(P_{prim},P_{opt})}(TG) := \{\ tg_i \in TG \mid tg_i.props() \subseteq \{P_{all} \cup P_{opt}\}\ ,\ (tg_i.props() \cap P_{prim}) \neq \emptyset\}$$

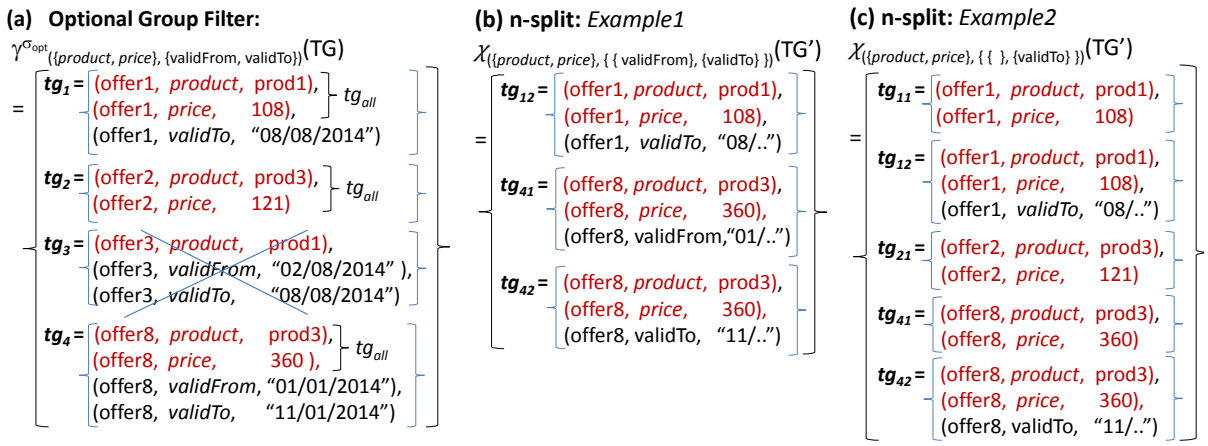


Figure 5.5: NTGA logical operators to evaluate composite graph patterns

Essentially, $\sigma^{\gamma_{opt}}$ ensures that triplegroups contain a matching triple for each of the properties in P_{prim}, and may contain matching triples for any of the properties in P_{opt}. For example, given $P_{prim} = \{product, price\}$, triplegroup tg_1, tg_2, and tg_4 are valid results for the $\sigma^{\gamma_{opt}}$ expression in Figure 5.5(a). However, tg_3 does not contain a matching triple for the required property *price*, and hence gets filtered out. Also, note that the valid triplegroups may have triples matching zero or more of the two optional properties $P_{opt} = \{validFrom, validTo\}$.

Retrieving Matches to Original Star Patterns. In order to retrieve valid triplegroups that match the original star patterns in the query, we need to extract subsets of triples in a triplegroup tg that are an exact match to one of the required pattern combinations. We define valid substructures based on *primary* (overlapping) properties that represent the common substructures in the original graph patterns, and *secondary* properties that represent the non-overlapping substructures. We now define an operator to extract the valid pattern combinations.

Definition 5.3.5 ***(n-split)*** *Given a set of triplegroups TG, a set of primary properties P_{prim}, and n sets of secondary properties $\{P_{sec_1}, P_{sec_2}, ..., P_{sec_n}\}$, the* ***n-split*** *operator χ creates a set of n triplegroups, where any triplegroup tg_i matches the star pattern $Stp_{o_i} = \{P_{prim}, P_{sec_i}\}$. Specifically,*

$$\chi_{(P_{prim},\{P_{sec_1},P_{sec_2},...,P_{sec_n}\})}(TG) := \{\ tg'_i = \{tg_{prim} \cup tg_{sec_i}\},\ i \in [1,n]\}$$

such that:

- $tg_{prim}, tg_{sec_i} \subseteq tg,\ tg \in TG$
- $tg_{prim}.props() = P_{prim}$ and $tg_{sec_i}.props() = P_{sec_i}$

In other words, the n-split operator extracts n subsets of triples in a triplegroup tg that match the different pattern combinations of a star pattern with n sets of secondary properties. Fig-

ure 5.5(b) shows triplegroups resulting from an n-split operation on TG', with primary properties $P_{prim} = \{product, price\}$, and two sets of secondary properties – $P_{sec_1} = \{validFrom\}$, and $P_{sec_2} = \{validTo\}$, respectively. While triplegroups tg_{41} conforms to the first pattern combination with properties { *product*, *price*, *validFrom* }, triplegroups tg_{12} and tg_{42} match the second combination { *product*, *price*, *validTo* }. Figure 5.5(c) shows another example of the n-split operation on TG' with the same set of primary properties, but with different sets of secondary properties – $P_{sec_1} = \{\}$, and $P_{sec_2} = \{validTo\}$, respectively. This means that the first pattern combination only contains primary properties (no secondary properties). Triplegroups tg_{11}, tg_{21}, $tg_{41} \in TG_{\{product,price\}}$, while triplegroups tg_{12}, $tg_{42} \in TG_{\{product,price,validTo\}}$.

Computing Joins between Composite Star Patterns. Processing joins involving composite star subpatterns may result in pattern combinations that are not relevant to the final result. For example, consider the graph patterns GP_{abcde} and GP_{abdef} with the following star pattern combinations:

$$GP_{abcde}\text{: } (Stp_{abc} \bowtie Stp_{de}) \text{ and } GP_{abdef}\text{: } (Stp_{ab} \bowtie Stp_{def})$$

The composite star subpatterns are $Stp_{ab\underline{c}}$ and $Stp_{de\underline{f}}$. Computing the join between such star patterns with optional properties, i.e., $(Stp_{ab\underline{c}} \bowtie Stp_{de\underline{f}})$ requires special handling of optional properties while producing the joined results. Additionally, such joins may result in pattern combinations such as *abde* that do not match either of the original patterns, GP_{abcde} or GP_{abdef}. Materialization of such irrelevant pattern combinations should be avoided as early as possible during the execution. We introduce a special triplegroup join operator to efficiently join composite subpatterns, by specifying valid pattern combinations using α conditions. We define an α condition as a set of structural constraints on a TG equivalence class, based on its secondary properties. In our example to ensure pattern combinations *abcde*, an α condition can be used to ensure that the triplegroups in the equivalence class $TG_{ab\underline{c}}$ contain at least one triple with property c, i.e., $tg.\texttt{props}(c) \neq \emptyset$. For brevity, we represent such a constraint as α: $c \neq \emptyset$.

Definition 5.3.6 *(α-**Join**) Let TG_x and TG_y be two triplegroup equivalence classes that join on variables jv_x and jv_y belonging to joining triple patterns tp_x and tp_y, respectively. Let $\delta(jv_x)$ denote a variable substitution in the triple matching tp_x. Let α_1, α_2,...,α_m be m conditions involving secondary properties in the equivalence classes. Then the α-**Join** operator $\bowtie^{\gamma}_{\{\alpha_1 \vee \ldots \vee \alpha_m\}}$ creates a joined triplegroup involving $tg_x \in TG_x$ and $tg_y \in TG_y$ if the following conditions hold:*

- Triplegroup tg_x contains a triple matching tp_x, and triplegroup tg_y contains a matching triple for tp_y, such that their variable substitutions match, i.e., $\delta(jv_x) = \delta(jv_y)$.
- tg_x and tg_y satisfy at least one of the α conditions.

Table 5.1: Computation of composite graph patterns using α-Join

	GP1		**GP2**		**GP'**		$\ltimes^{\gamma}_{(\alpha_1 \vee \alpha_2)}$(jv_x:$TG_{Stp'_a}$, jv_y:$TG_{Stp'_b}$)	
	Stp_a	Stp_b	Stp_α	Stp_β	Stp'_a	Stp'_b	α_1	α_2
(1)	ab	de	ab	de	ab	de	–	–
(2)	ab	de	ab	def	ab	$de\underline{f}$	$f = \emptyset$	$f \neq \emptyset$
(3)	ab	de	abc	def	$ab\underline{c}$	$de\underline{f}$	$c = \emptyset \wedge f = \emptyset$	$c \neq \emptyset \wedge f \neq \emptyset$
(4)	abc	de	ab	def	$ab\underline{c}$	$de\underline{f}$	$c \neq \emptyset \wedge f = \emptyset$	$c = \emptyset \wedge f \neq \emptyset$
(5)	abc	de	ab	$defg$	$ab\underline{c}$	$de\underline{\underline{fg}}$	$c \neq \emptyset \wedge f = \emptyset \wedge g = \emptyset$	$c = \emptyset \wedge f \neq \emptyset \wedge g \neq \emptyset$

The basic idea is to join the triplegroups only if they form a valid pattern combination that is relevant to the original graph patterns in a query. Table 5.1 shows various examples of original graph patterns $GP1$ and $GP2$, their composite graph pattern GP', and the α constraints for the α-Join operator. For example, α conditions in row (4) ensure that pattern combinations such as $abde$ and $abcdef$ are not produced. Similarly, conditions α_1 and α_2 in row (5) correspond to the original patterns $abcde$ and $abdefg$ respectively, hence avoiding materialization of triplegroups matching irrelevant patterns such as $abde$, $abdef$, $abdeg$, $abcdef$, $abcdeg$, $abcdefg$, etc.

Computing Grouping-Aggregations on Composite Graph Patterns. In order to evaluate RDF graph analytical queries with multiple grouping and aggregation phases, we define an operator in the same spirit as the relational *MD-Join* operator. In additional to processing grouping and aggregations on TG equivalence classes based on a composite graph pattern, the new operator should apply the grouping and aggregations only on triplegroups that satisfy the associated α condition.

Definition 5.3.7 ***(TG MD-Join)*** *Let TG_{base} and TG_{detail} be two triplegroup equivalence classes, θ be a condition involving variable substitutions in TG_{base} and TG_{detail}, and let l be a list of aggregation functions (f_1, f_2,...,f_m) over aggregation variables a_1, a_2, ..., a_m, respectively. Let α be a condition involving secondary properties in TG_{detail}. Then the triplegroup MD-Join operator γ^{MDJ}(TG_{base}, TG_{detail}, l, θ, α) creates a set of aggregated triplegroups ATG, where any aggregated triplegroup $agtg_i \in ATG$ satisfies the following conditions:*

- Each base triplegroup $btg_i \in TG_{base}$ is associated with a set of triplegroups in TG_{detail}, using the following function :

  $$RNG(btg_i, TG_{detail}, \theta, \alpha) = \{\ dtg \in TG_{detail}\ \}$$

  such that triplegroups btg_i and dtg satisfy conditions in θ and α.

- For each base triplegroup $btg_i \in TG_{base}$, there exists an aggregated triplegroup $agtg_i \in ATG$ with triples $t_{ik} \in agtg_i$ that contain values corresponding to some aggregation function f_k and variable a_k such that :

$$t_{ik} = (grpKey,\ \texttt{createProp}(f_k, a_k),\ f_k\_agtg_i\_a_k)$$

whose values are computed as follows :

- $grpKey$ = subject of btg_i, represents the grouping key.
- function $\texttt{createProp}(f_k, a_k)$ returns a unique property based on combination of aggregation function and variable.
- Aggregate value $f_k\_agtg_i\_a_k$ is computed by applying the aggregation function f_k on variable substitutions of a_k in triplegroups matching $RNG(btg_i,\ TG_{detail},\ \theta,\ \alpha)$.

Each base triplegroup $btg_i \in TG_{base}$ corresponds to a distinct grouping key and produces an aggregated triplegroup $agtg_i \in ATG$. The subset of triplegroups in TG_{detail} that contribute to an aggregated triplegroup $agtg_i$ is computed using the function $RNG(btg_i,\ TG_{detail},\ \theta,\ \alpha)$, that returns the set of triplegroups in TG_{detail} that satisfy the join condition θ as well as the α condition with respect to the base triplegroup btg_i. The θ condition defines the join condition between the base triplegroup and triplegroups in TG_{detail}, while the α condition defines the restrictions based on secondary properties in TG_{detail}. Further, each aggregated triplegroup $agtg_i \in ATG$ contains a triple corresponding to each of the aggregation conditions.

Consider an example TG *MD-Join* operation between a base TG equivalence class TG_{Base} and a detail TG equivalence class $TG_{\{ty18,\underline{pf},pr,pc,ve,cn\}}$ to compute the groupings based on ProductFeature and Country, and SUM and COUNT aggregations on price. The resultant aggregated triplegroups are represented in Figure 5.6.

The θ condition computes the RNG of a base triplegroup based on the value bindings of the grouping variables $?feature$ and $?country$ in TG_{detail}. For example triplegroup dtg_1, the value bindings $\delta_1(?feature)$={Feat1} and $\delta_1(?country)$={UK}. The α condition $pf \neq \emptyset$ ensures the presence of a triple with secondary property pf corresponding to the product feature. In our example, triplegroup dtg_2 does not satisfy the α condition and hence does not contribute to any of the aggregated triplegroups. The RNG for the example base triplegroups in TG_{Base} can be computed as follows:

$RNG(btg_1,\ TG_{\{ty18,\underline{pf},pr,pc,ve,cn\}},\ \theta,\ \alpha) = \{\ dtg_1, dtg_4\}$
$RNG(btg_2,\ TG_{\{ty18,\underline{pf},pr,pc,ve,cn\}},\ \theta,\ \alpha) = \{\ dtg_3\ \}$
$RNG(btg_3,\ TG_{\{ty18,\underline{pf},pr,pc,ve,cn\}},\ \theta,\ \alpha) = \emptyset$
$RNG(btg_4,\ TG_{\{ty18,\underline{pf},pr,pc,ve,cn\}},\ \theta,\ \alpha) = \{\ dtg_3\ \}$

For any base triplegroup btg_i, the aggregated triplegroup is computed by aggregating the subset of triplegroups in TG_{detail} that satisfy the RNG of btg. For example, aggregated triplegroup $agtg_1$ is computed by aggregating triplegroups dtg_1 and dtg_4, that satisfy the RNG

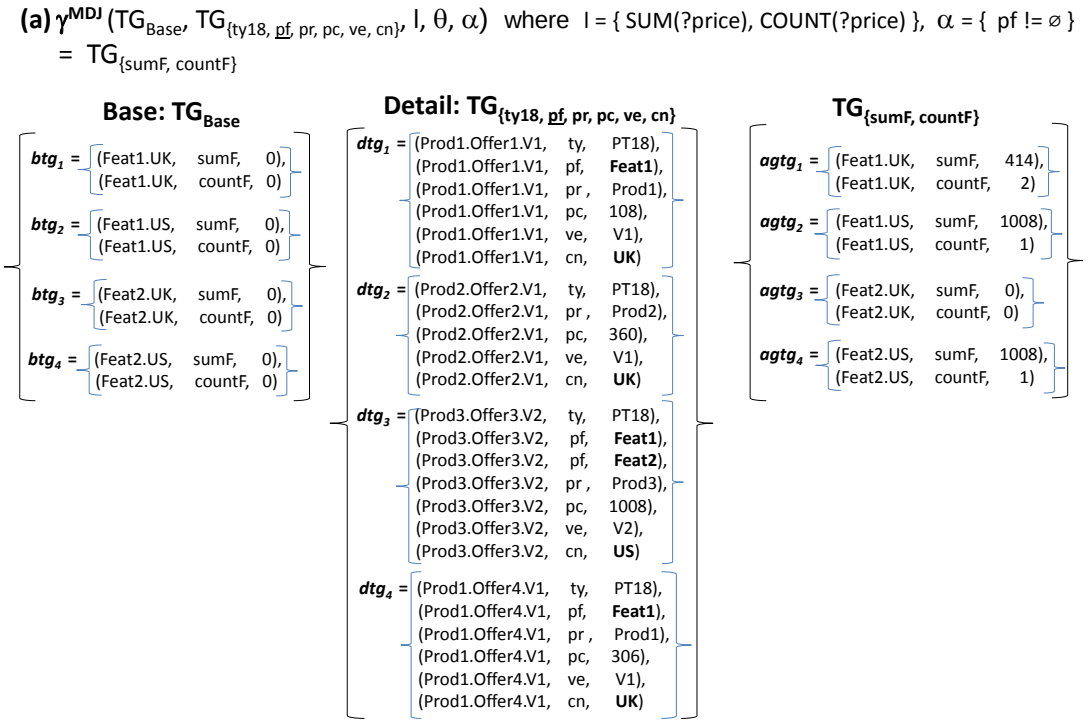


Figure 5.6: Example TG *MD-Join* operation between base TG equivalence class TG_{Base} and detail TG equivalence class $TG_{\{ty18,pf,pr,pc,ve,cn\}}$

function of btg_1. Additionally, since the RNG function of base triplegroup btg_3 returns an empty set, the corresponding aggregated triplegroup $agtg_3$ retains the default values in btg_3.

5.4 Query Execution

5.4.1 Translation to MapReduce Plans

The logical operators proposed in the previous section are integrated into RAPID+ [51]. As with other relational-style Hadoop extensions, the query compilation process in RAPID+ begins with a logical plan, which is compiled into a plan with physical operators. A physical operator is either a single function or a function pair that corresponds to map and reduce phases of the logical operator. For example, the optional group filtering operator `TG_OptGrpFilter` ($\sigma^{\gamma_{opt}}$) is a single function and can be pipelined with other operators in either the map or the reduce phase. However, operators such as triplegroup join `TG_Join` and triplegroup MD-Join `TG_MDJ` which require redistribution of input, are defined as function-pairs. The assignment of the physical operators to MapReduce cycles, constitutes a MapReduce plan.

The MapReduce plan for an RDF analytical query, first executes the optional group-filtering operator `TG_OptGrpFilter` on the subject triplegroups, to produce triplegroups that match the

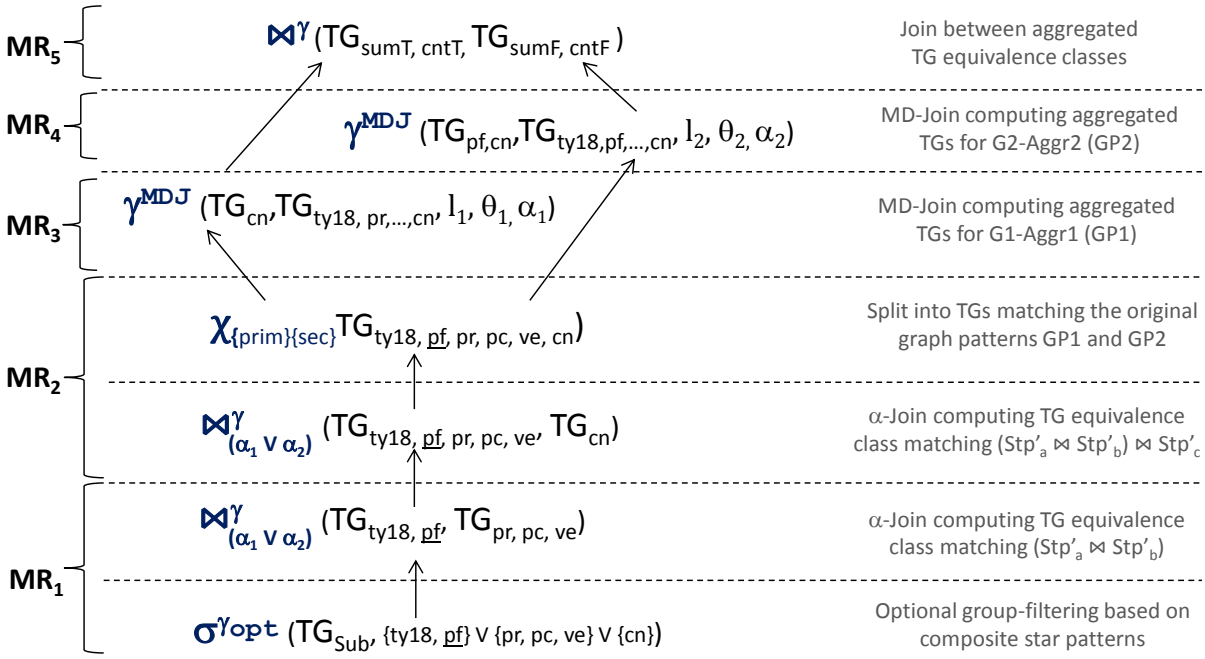


Figure 5.7: NTGA-based MR execution plan: Evaluating composite graph pattern GP'
Prefixes: $ty18(typePT18)$, $pf(prodFeature)$, $pr(product)$, $pc(price)$, $ve(vendor)$, $cn(country)$

composite star subpatterns in the query. The `TG_OptGrpFilter` operator can either be executed in the reduce of `TG_GroupBY` that produces the subject triplegroups or can be executed in the initial map phase if the subject triplegroups are created using a pre-processing phase. If there are no overlapping star subpatterns in the input query or if the query contains a single graph pattern, perfect triplegroups are produced using the basic `TG_OptGrpFilter` operator ($P_{opt} = \emptyset$). The joins between triplegroups that match the subpatterns are computed using NTGA's `TG_Join` operator in the subsequent MR cycles. This is followed by additional MR cycles that execute one or more `TG_MDJ` operators, to compute the required grouping and aggregations. We illustrate the case of overlapping graph patterns using the running example in this chapter.

NTGA-based Evaluation for Query 1. The graph patterns $GP1$ and $GP2$ in example 1 can be re-written as the following composite graph pattern:

$$GP'\text{: } Stp_{\{ty18,\underline{pf}\}} \bowtie Stp_{\{pr,pc,ve\}} \bowtie Stp_{cn}$$

Let TG_{Sub} be a set of subject triplegroups created using a `TG_GroupBy` on the Subject column of the input triples relation T. Figure 5.7 shows the NTGA query plan for example 1, along with the assignment of the operators to the MR cycles. First, NTGA's `TG_OptGrpFilter` operator ($\sigma^{\gamma_{opt}}$) is executed in the map phase of cycle MR_1, which creates three sets of triplegroup equivalence classes, $TG_{\{ty18,\underline{pf}\}}$, $TG_{\{pr,pc,ve\}}$, and TG_{cn}, that match the three star subpatterns

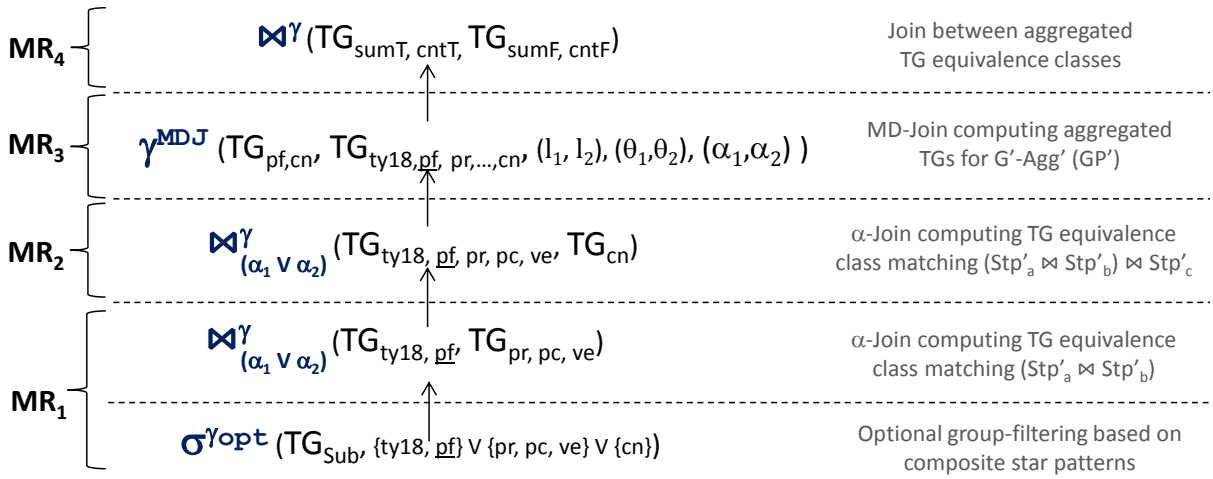


Figure 5.8: NTGA-based MR execution plan: Evaluating MD-Join on composite graph pattern GP' when aggregations can be computed independently

in the composite graph pattern. The first α-Join operator, $\bowtie^{\gamma}_{(\alpha_1 \vee \alpha_2)}$, also executes in map-reduce phases of MR_1, and computes the join between equivalence classes $TG_{\{ty18,\underline{pf}\}}$ and $TG_{\{pr,pc,ve\}}$, respectively, with condition α_1:$(pf = \emptyset)$, and α_2:$(pf = \emptyset)$. The result of this join is the triplegroup equivalence class $TG_{\{ty18,\underline{pf},pr,pc,ve\}}$, which is then joined with the equivalence class TG_{cn} using another α-Join operator in map-reduce phases of cycle MR_2. The result of this join is the triplegroup equivalence class $TG_{\{ty18,\underline{pf},pr,pc,ve,cn\}}$, which represents the triplegroups matching the composite graph pattern. Next, the *n-split* operator is executed in the reduce of cycle MR_2 to extract the equivalence classes $TG_{\{ty18,pr,pc,ne,cn\}}$ and $TG_{\{ty18,pf,pr,pc,ne,cn\}}$, that match the original graph patterns $GP1$ and $GP2$, respectively. Subsequent MR cycle MR_3 executes the triplegroup MD-Join operator (γ^{MDJ}) on the equivalence class $TG_{\{ty18,pr,pc,ne,cn\}}$, which computes the sum and count of product prices per country. This is followed by cycle MR_4 that executes the triplegroup MD-Join operator on the equivalence class $TG_{\{ty18,pf,pr,pc,ne,cn\}}$, to compute the sum and count of product prices per feature-country combination. Once the two aggregations are computed, the final ratio is computed using a map-only phase to join the two aggregated triplegroup equivalence classes.

Parallel Execution of `TG_MDJs`. One of the useful optimizations provided in [11, 26] is that, a series of two *MD-Join* operations can be combined into a single generalized *MD-Join* if, (i) they involve the same detail relation, and (ii) they are are independent, i.e., the θ conditions of the second *MD-Join* does not involve augmented columns generated by the first *MD-Join*. This optimization can be applied to our example query if we apply the triplegroup *MD-Join* operator on the composite equivalence class $TG_{ty18,\underline{pf},pr,pc,ne,cn}$. Figure 5.8 shows the NTGA query plan and MapReduce execution plan that enables parallel execution of the triplegroup

Algorithm 7: MR job workflow in RDFAnalytics

Job_i: α-Join between TG equivalence classes
Map:
1 $TG' \leftarrow$ `TG_OptGrpFilter`(TG, $<EC, \{P_{prim}, P_{opt}\}>$);
2 `TG_AlphaJoin`(TG').Map();
Reduce:
3 $TG'' \leftarrow$ `TG_AlphaJoin`(TG').Reduce();
Job_k: Multi-dimensional Join on TG equivalence classes
Map:
4 `TG_MDJ`(TG'').Map();
Reduce:
5 $AggTG \leftarrow$ `TG_MDJ`(TG'').Reduce();
Job_n: Join on Base-values TG equivalence classes
Map:
6 `TG_AggJoin`($mdj\#grpKey$, $AggTG$);

MD-Join operator (γ^{MDJ}) by combining them as a generalized operator:

$$\gamma^{MDJ}(\ TG_{Base},\ TG_{\{ty18,\underline{pf},pr,pc,ne,cn\}},\ (l_1,\ l_2),\ (\theta_1,\ \theta_2),\ (\alpha_1,\ \alpha_2)\ )$$

which is executed in MR cycle MR_3. In the case of the traditional generalized *MD-Join* operation, the aggregations l_1 and l_2 are updated concurrently in the base-values relation, by a single scan of the detail relation. However, in the case of the `TG_MDJ` operator executed across map-reduce phases, the aggregated triplegroups corresponding to the two `TG_MDJ` operations may be spread across different reducers. Hence, we require a subsequent map-only cycle to join the aggregated triplegroups and compute the final ratios based on the aggregated triplegroups.

5.4.2 Algorithms for Physical Operators

Algorithm 7 gives an overview of the job flow for the key phases in NTGA-based evaluation of RDF Graph analytical queries – Job_i, that computes the join between the triplegroup equivalence classes, and Job_k, that computes the multi-dimensional join between the triplegroup equivalence classes. If there is a structural overlap in the input graph patterns, the triplegroup equivalence classes are computed based on the *composite* graph pattern. This is achieved by evaluating the optional group-filtering operator, `TG_OptGrpFilter`, based on the required and optional properties in the composite graph pattern.

Algorithm 8 shows the pseudocode for the `TG_OptGrpFilter` operator. The $(Property, Object)$ pairs in a triplegroup ($tempMap$ in line 1), are matched with all the equivalence classes (star patterns) in the query (line 2). For each matched equivalence class EC, the triplegroup is considered as a match only if it contains all the required properties P_{all} (lines 4-9). The required properties P_{prim} are extracted from the triplegroup (line 5). If the triplegroup contains any of the optional properties P_{opt}, they are also extracted and added to $propMap$ (lines 6-8). In summary, the resultant annotated triplegroup contains all the required properties, and may

Algorithm 8: TG_OptGrpFilter

```
   optGrpFilter (tg, ECList: ⟨EC, {P_prim, P_opt}⟩)
1  tempMap ← extract triples in tg;
2  matchedECList ← match(tempMap, ECList);
3  foreach EC ∈ matchedECList do
4      if (EC.P_all ⊆ tempMap.keySet) then
5          propMap ← extract P_prim entries from tempMap;
6          foreach optProp ∈ EC.P_opt do
7              if tempMap contains optProp then
8                  Add optProp entry from tempMap to propMap;
9          emit ⟨ AnnTG(Sub, EC, propMap) ⟩
```

contain one or more of the optional properties. Triplegroups that do not contain any of the required properties, are pruned out at this stage.

***Job*$_i$:α-Join between TG equivalence classes.** The input to this phase is a set of annotated triplegroups, belonging to two equivalence classes (that match some composite subpattern) whose join is to be computed. The output is a set of annotated triplegroups, representing the joined result between the equivalence classes. In order to eliminate pattern combinations that do not match any of the original graph patterns, all the valid combinations are encoded as a list of α conditions, one for each of the original graph patterns. Algorithm 9 shows the map-reduce functions for the `TG_AlphaJoin` operator that integrates α-based filtering of irrelevant triplegroups during the join between equivalence classes.

Algorithm 9: TG_AlphaJoin

```
   Map (key:null, val: AnnTG atg)
1  if join on Sub then
2      emit ⟨ atg.Sub, atg ⟩;
   else if join on Obj then
3      objList ← extract objects corr. to join property from atg;
4      foreach obj ∈ objList do
5          emit ⟨ obj, atg ⟩;
   Reduce (key:joinKey, val:List of AnnTGs TG') ;
6  alphaList < α_1, ..., α_n > ← α restrictions for current join;
7  leftList ← extract leftEC AnnTGs from TG';
8  rightList ← extract rightEC AnnTGs from TG';
9  foreach ltg ∈ leftList do
10     foreach rtg ∈ rightList do
11         if ∃ α ∈ alphaList s.t. ltg and rtg satisfy α then
12             emit ⟨ joinTGs(ltg, rtg)⟩;
```

In the map phase, the input triplegroup *atg* is tagged (lines 1-5) either on the Subject or Object value, based on the type of join. Since NTGA implicitly represents subgraphs with

multi-valued properties, unnesting of a triplegroup is necessary when the join involves a multi-valued property (lines 3-5). In the reduce phase, each *reduce*() receives AnnTGs corresponding to the same join key. The *alphaList* (line 6) contains the n α conditions that correspond to the n original graph patterns. The algorithm iterates through the triplegroups belonging to the left and the right equivalence classes ($leftEC$ and $rightEC$ in lines 7-8), and computes the join only if at least one of the α conditions is satisfied (line 11). In other words, a pair of triplegroups that do not match any of the valid join combinations encoded in the α list, are eliminated. For example, if TG_{abc} and TG_{def} are the left and right equivalence classes, and the valid join combinations are *abcde* and $abdef$, then the α conditions restrict the join between triplegroups, i.e., α_1=$\{c{\neq}\emptyset \wedge f{=}\emptyset\}$, α_2=$\{c{=}\emptyset \wedge f{\neq}\emptyset\}$. Thus, a left triplegroup *ltg* with properties *ab* and a right triplegroup *rtg* with properties *de* do not satisfy either of the α conditions, and hence are not joined.

Job_k **: Multi-dimensional Join on TG equivalence classes.** The input to this phase is a set of annotated triplegroups that match the composite graph pattern (or one of the original graph patterns). The output is a set of aggregated triplegroups that contain the required aggregations.

Algorithm 10 shows the map-reduce functions for the `TG_MDJ` operator. In order to reduce the number of intermediate triplegroups that are shuffled to the reducers, we implement a hash-based aggregation per mapper, i.e., instead of generating a map output for each map input triplegroup, we partially aggregate the triplegroups at each mapper. The triplegroups are aggregated into a hashmap $multiAggrMap$ that is accessible across different map() invocations at a mapper. This hash-based aggregation resembles a local combiner within each mapper.

One or more multi-dimensional joins may be evaluated in the same job. The list of multi-dimensional joins to be executed in the current job are available in $mdjList$ (line 2). Each multi-dimensional join *mdj* (identified using *id*) contains a θ condition, from which the grouping key is extracted (line 4). Each *mdj* also contains the list of aggregations to be computed, and the equivalence class relevant to the current *mdj*. In the map phase, as each input triplegroup *atg* is processed, the relevant multi-dimensional joins is applied if the associated α condition is satisfied (lines 3-16). The aggregation list contains a list of aggregations *aggregate*, with specifications about the type of aggregation, e.g., SUM, COUNT etc., and the attribute on which to aggregations, e.g., aggregate based on the subject of *atg* (lines 7-8) or aggregate based on the object of a specific property such as *price* (line 9). The computed aggregate value is stored in $currVal$. A new property is created based on the type and attribute of the aggregate ($aggPropName$ in line 10). The aggregated result is stored as a (Property, Object) pair, i.e., ($aggPropName$, $currVal$), and added to an aggregated triplegroup $currAggTg$ (line 11). One all the aggregations related to the current *mdj* are computed, the aggregated triplegroup $currAggTg$ is aggregated with existing values in the mapper's global hashmap $multiAggrMap$

Algorithm 10: TG_MDJ

```
   Map (key:null, val: AnnTG atg)
 1 Initialize multiAggrMap for hash-based Map() aggregation
 2 foreach mdj< id, aggrList, theta, alpha > ∈ mdjList do
 3     if atg satisfies alpha then
 4         grpKey ← extract mdj.theta from atg ;
 5         aggrList ← mdj.aggrList;
 6         foreach aggregate < type, attrib > ∈ aggrList do
 7             if aggregate on Sub then
 8                 currVal ← apply aggregate.type to atg.getSub();
               else
 9                 currVal ← apply aggregate.type to atg.getProp(aggregate.attrib);
10             aggPropName ← createNewProp(aggregate.type, aggregate.attrib);
11             Add (aggPropName, currVal) to currAggTg ;
12         if multiAggrMap.containsKey(mdj.id#grpKey) then
13             oldAggTg ← multiAggrMap.get(mdj.id#grpKey);
14             newAggTg ← aggregate(oldAggTg, currAggTg, aggrList);
15             Update (mdj.id#grpKey, newAggTg) in multiAggrMap;
           else
16             Add (mdj.id#grpKey, currAggTg) to multiAggrMap;
   Map.clean ()
17 foreach (mdjId#grpKey) ∈ multiAggrMap.keySet() do
18     aggTg ← multiAggrMap.get(mdjId#grpKey);
19     emit ⟨ mdjId#grpKey, aggTg⟩ ;
   Reduce (key:mdjId#grpKey, val:List of AggTGs TG′) ;
20 aggrList ← mdjList.get(mdjId).getAggrList();
21 foreach aggTg ∈ TG′ do
22     grpAggrTg ← aggregate(grpAggrTg, aggTg, aggrList);
23 emit ⟨ mdjId#grpKey, grpAggrTg⟩;
```

that correspond to the same *mdj-grpKey* combination (12-16). Once the map() functions are complete, we loop through the entries in the global hashmap *multiAggrMap* and output the pre-aggregated triplegroups (lines 17-19). In the reduce phase, each reduce() receives pre-aggregated triplegroups corresponding to the same *mdj-grpKey* combination. The triplegroups are further aggregated (lines 21-23) and the output triplegroups are written onto the HDFS.

Note on `TG_AggJoin`**.** Multi-phase aggregation queries require joining of outputs of the *MD-Join* operations. In the case of graph patterns with no structural overlap, each *MD-Join* is executed as a separate MapReduce job. Hence, the aggregated triplegroups from the different `TG_MDJ` operators need to be joined. In the case of structural overlap of all graph patterns in the query, a single `TG_MDJ` operator executes all independent grouping-aggregations. However, the different *MD-Join* results are distributed across reducers, and thus need to be joined. Hence, we implement a special map-only operator that replicates the smaller aggregated TG equivalence class to all mappers, and joins with the larger aggregated TG equivalence class. In the case that a particular *MD-Join* contains a grouping on `ALL`, the corresponding aggregated

TG equivalence class is replicated across mappers. This is because aggregated triplegroups corresponding the group key `ALL`, need to be joined with all triplegroups from the second aggregated TG equivalence class.

5.5 Empirical Evaluation

This section presents a comprehensive evaluation of the proposed algebraic optimizations for simple (single grouping) as well as multi-grouping analytical queries on RDF graphs. In the rest of this section, RAPID+ optimized for RDF analytical queries is denoted as *RAPIDAnalytics*. The performance of *RAPIDAnalytics* was compared with Hive, a popular Hadoop-based platform with relation-style query processing and optimization. For multi-grouping queries, two Hive-based approaches were included, (i) *Hive (Naive)* – the original SPARQL query with multiple graph patterns was translated into HiveQL, and (ii) *Hive (MQO)* – a MQO-based rewriting [57] of multiple graph patterns was translated into HiveQL using left outer joins, followed by a second HiveQL query to compute the appropriate groupings and aggregations. Additionally, performance was compared with *RAPID+ (Naive)*, that sequentially evaluates the multiple graph patterns and the grouping-aggregation phases.

5.5.1 Setup

Experiments were conducted on NCSU's VCL [83], where each node in the cluster was a dual core Intel X86 machine with 2.33GHz processor speed, 4GB memory and running Red Hat Linux. 10, 50, and 60-node Hadoop clusters (block size set to 128MB, 1GB heap-size for child jvms) were used with Hive release 0.12.0 and Hadoop 0.20.2.

Testbed - Dataset and Queries: Experiments were conducted using two synthetic benchmark datasets and one real-world dataset. The two synthetic datasets were generated by the Berlin SPARQL Benchmark (BSBM) [22] data generator tool - *BSBM-500K* (43GB dataset with 500K Products, total ∼175 million triples) and *BSBM-2M* (172GB dataset with 2 million Products, total ∼700 million triples). Evaluation of real-world RDF analytical queries was conducted on a chemogenomics RDF data warehouse, *Chem2Bio2RDF* [31], that is an aggregation of data from multiple chemical, biological, and chemogenomics data sources that link chemical compounds with targets, genes, side effects, diseases, and publications (60GB dataset with total ∼340 million triples). Additional experiments were conducted on *PubMed* dataset (*Bio2RDF* release 2) [18] containing ∼1.7 billion triples (230GB dataset).

The evaluation tested simple RDF analytical queries ($G1$-$G9$) as well as multi-grouping queries ($MG1$-$MG10$) with varying selectivities (low vs. high), varying granularity of groupings (`GROUP BY` ALL vs. `GROUP BY` feature vs. `GROUP BY` country-feature combinations), and varying

Table 5.2: Testbed Queries: Structures of Evaluated RDF Analytical Queries (#TPs is number of triple patterns in a graph pattern)

RDF Analytical Queries with Single Grouping-Aggregation					
Dataset	**Query**	**GP1 (# TPs)**	**GROUP BY**	**GP2 (# TPs)**	**GROUP BY**
BSBM	G1:lo	Stp_1:2, Stp_2:2	ALL	-	-
BSBM	G2:hi	Stp_1:2, Stp_2:2	ALL	-	-
BSBM	G3:lo	Stp_1:3, Stp_2:2	ProductFeature	-	-
BSBM	G4:hi	Stp_1:3, Stp_2:2	ProductFeature	-	-
Chem2Bio	G5	Stp_1:4, Stp_2:2, Stp_3:2, Stp_4:1	CompoundId	-	-
Chem2Bio	G6	Stp_1:4, Stp_2:1, Stp_3:2	CompoundId	-	-
Chem2Bio	G7	Stp_1:2, Stp_2:1, Stp_3:2, Stp_4:2	PathwayId	-	-
Chem2Bio	G8	Stp_1:4, Stp_2:2, Stp_3:2, Stp_4:1	CompoundId	-	-
Chem2Bio	G9	Stp_1:1, Stp_2:2	GeneSymbol	-	-
RDF Analytical Queries with Multiple Grouping-Aggregations					
Dataset	**Query**	**GP1 (# TPs)**	**GROUP BY**	**GP2 (# TPs)**	**GROUP BY**
BSBM	MG1:lo	Stp_1:3, Stp_2:2	ProductFeature	Stp_1:2, Stp_2:2	ALL
BSBM	MG2:hi	Stp_1:3, Stp_2:2	ProductFeature	Stp_1:2, Stp_2:2	ALL
BSBM	MG3:lo	Stp_1:3, Stp_2:3, Stp_3:1	ProductFeature, Country	Stp_1:2, Stp_2:3, Stp_3:1	Country
BSBM	MG4:hi	Stp_1:3, Stp_2:3, Stp_3:1	ProductFeature, Country	Stp_1:2, Stp_2:3, Stp_3:1	Country
Chem2Bio	MG6	Stp_1:4, Stp_2:2, Stp_3:2	CompoundId, GeneSymbol	Stp_1:4, Stp_2:2, Stp_3:2	CompoundId
Chem2Bio	MG7	Stp_1:4, Stp_2:2, Stp_3:2	CompoundId, Drug	Stp_1:4, Stp_2:2, Stp_3:2	CompoundId
Chem2Bio	MG8	Stp_1:4, Stp_2:2, Stp_3:2	CompoundId, GeneSymbol	Stp_1:4, Stp_2:2, Stp_3:2	ALL
Chem2Bio	MG9	Stp_1:2, Stp_2:1	GeneSymbol	Stp_1:2, Stp_2:1	ALL
Chem2Bio	MG10	Stp_1:3, Stp_2:1	Disease, GeneSymbol	Stp_1:2, Stp_2:1	GeneSymbol
PubMed	MG11	Stp_1:2, Stp_2:2	Country	Stp_1:2, Stp_2:1	ALL
PubMed	MG12	Stp_1:2, Stp_2:2	PublicationType Country	Stp_1:1, Stp_2:2	Country
PubMed	MG13	Stp_1:3, Stp_2:1	Author PublicationType	Stp_1:3, Stp_2:1	PublicationType
PubMed	MG14	Stp_1:3, Stp_2:1	Author PublicationType	Stp_1:3, Stp_2:1	PublicationType
PubMed	MG15:lo	Stp_1:3, Stp_2:1	AuthorLastName	Stp_1:3, Stp_2:1	ALL
PubMed	MG16:hi	Stp_1:3, Stp_2:1	AuthorLastName	Stp_1:3, Stp_2:1	ALL
PubMed	MG17	Stp_1:3, Stp_2:2	Country	Stp_1:3, Stp_2:1	ALL
PubMed	MG18	Stp_1:3, Stp_2:2	Author Country	Stp_1:2, Stp_2:2	Country

structures of associated graph patterns. A summary of the query structures of the testbed queries are represented in Table 5.2. Queries $G1$-$G4$ and $MG1$-$MG4$ were adapted from the Berlin SPARQL Benchmark Business Intelligence Use Case 3.1 [2], and included queries that are built around the e-commerce use case representing analytical questions by vendors, customers, and owners of the e-commerce portal. Queries $G5$-$G9$ and $MG6$-$MG10$ were adapted based on case studies [31] on the Chem2Bio2RDF dataset with use cases such as disease-specific drug discovery and identification of adverse drug reactions. For example, query $G5$ retrieves drug-like compounds in the PubChem BioAssay dataset that share at least two targets with a drug of interest (Dexamethasone) in the DrugBank. Such a query enables identification of drug-like compounds that show similar polypharmacology (interaction with multiple targets) with a well known marketed drug. Query $G6$ finds compounds in the PubChem dataset that are active against at least two target proteins that are associated with a disease-related biological pathway. Such a query is useful in drug discovery to identify potential multiple pathway inhibitors for *MAPK*, that is responsible for coordinating cell proliferation, differentiation, and death. The third query from the case study, query $G7$ retrieves KEGG pathways that contain at least two efficient gene targets that are associated with a given side-effect, such as hepatomegaly or swelling of liver. Such a query is useful to retrieve pathways that are associated with drug hepatotoxicity or liver toxicity. Queries $MG6$-$MG10$ were extended based on the basic case study scenarios. Queries $MG11$-$MG18$ involve aggregation on PubMed records. All evaluated queries in SPARQL are available in Appendix D.

Pre-processing Phase: As a first step, URIs of all triples were replaced with property prefixes to reduce the size of input data. For Hive-based approaches, triples were vertically partitioned (VP) based on properties and loaded into Hive tables with schema (Subject, Object) using multiple MapReduce jobs. For triples with *rdf:type*, property-object partitions were created and stored in Hive tables with schema (Subject). The BSBM dataset contains a small number of distinct property types (∼40 property types), but a large number of *rdf:type* Objects – *BSBM-500K* contains more than 3000 types of 'ProductTypes' such as ProductType1, ProductType2, etc. Hence, the VP-based pre-processing phase for a subset of VP relations took ∼6 hours for *BSBM-500K* (43GB dataset) on a 10-node cluster. *BSBM-2M* has larger number of ProductTypes and hence VP tables were created only for a subset of properties used in the evaluated queries. On the other hand, the Chem2Bio2RDF dataset contains 176 distinct property types and 27 distinct *rdf:type* Objects. The distribution of the number of triples varies greatly across properties. Table 5.3 summarizes the number of triples per VP table for the various properties used in the testbed queries, which is useful in analysing selectivity of the evaluated queries.

Note that all Hive tables were stored as *Optimized Row Columnar* (ORC) [5] file format with default compression settings. The ORC file format aggressively compresses the stored

Table 5.3: Data Statistics after Pre-processing in Hive: Number of triples and size of vertically-partitioned Hive table (ORC format with default compression technique)

***BSBM-2M* (172GB) – 700,623,319 triples (136,364,658 subject triplegroups)**					
Property Name	**No. of Triples**	**Size (MB)**	**Property Name**	**No. of Triples**	**Size (MB)**
rdfs:type-PT1	2,000,000	7.49	*rdfs:type-PT9*	127,975	0.77
rdfs:label	2,157,707	27.19	*bsbm:country*	1,084,121	4.18
bsbm:productFeature	38,571,526	87.61	*bsbm:product*	40,000,000	265.43
bsbm:price	40,000,000	235.07	*bsbm:vendor*	40,000,000	115.54

***Chem2Bio2RDF* (60GB) – 340,946,808 triples (54,164,628 subject triplegroups)**					
Property Name	**No. of Triples**	**Size (MB)**	**Property Name**	**No. of Triples**	**Size (MB)**
pubchem:CID	425,896	3.53	*pubchem:outcome*	190,583	0.51
pubchem:Score	190,583	0.69	*pubchem:gi*	190,583	0.51
uniprot:gi	331	0.01	*uniprot:geneSymbol*	31,228	0.22
drugbank:gene	10,091	0.05	*drugbank:DBID*	14,855	0.08
drugbank:Generic_Name	4,764	0.06	*drugbank:SwissProt_ID*	14,855	0.07
kegg:protein	21,190	0.06	*kegg:Pathway_name*	192	0.01
medline:side_effect	10,489,676	53.53	*medline:disease*	12,612,636	61.95
kegg:pathwayid	192	0.01	*medline:gene*	5,252,844	26.84
sider:side_effect	1,385	0.01	*sider:cid*	61,102	0.09

***PubMed* (230GB) – 1,688,043,771 triples (673,479,846 subject triplegroups)**					
Property Name	**No. of Triples**	**Size (MB)**	**Property Name**	**No. of Triples**	**Size (MB)**
pubmed:grant_agency	3,611,589	14.35	*pubmed:grant_country*	3,611,589	12.1
pubmed:mesh_heading	184,285,466	639.11	*pubmed:publication_type*	33,666,437	110.51
pubmed:author	63,728,799	290.19	*pubmed:chemical*	37,040,148	166.67
pubmed:last_name	64,617,934	408.59	*pubmed:journal*	17,794,848	102.40
pubmed:grant	3,611,589	21.20	*dct:title*	17,794,848	632.21

data, with almost ∼80-96% reduction in data size when compared to text file storage. Table 5.3 shows the size of Hive VP tables stored as ORC files. The ORC file format is known to provide high performance benefits to Hive, with optimizations such as light-weight indexes to skip row groups that do not pass predicate filtering, block-mode compression based on data types etc. Additionally, the ORC file format is advantageous for analytical queries since it can store column-level aggregates such as COUNT, SUM, MIN, etc.

In the case of *RAPIDAnalytics*, subject triplegroups were generated using a single MapReduce job by grouping the triples based on the subject column. The subject triplegroups were stored in separate text files based on their equivalence class (set of properties that define their structure). In order to reduce the number of equivalence class combinations, the associated *rdf:type* triples were stored in separate files. Further, to deal with the large number of 'ProductType's that would potentially generate large number of small files, *rdf:type* triples with

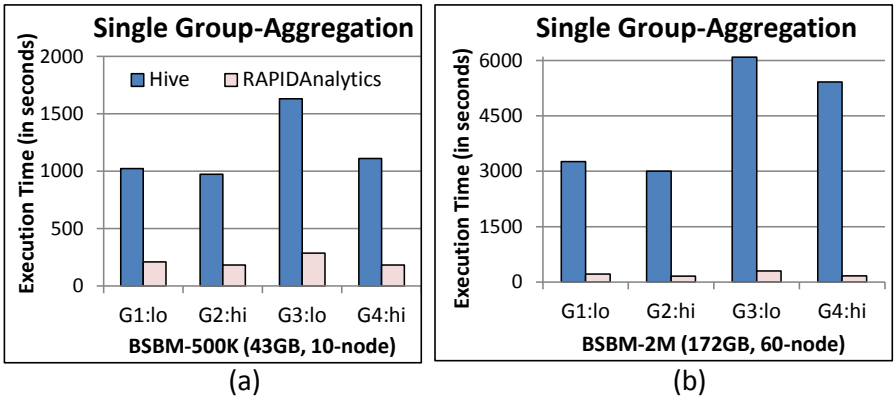


Figure 5.9: A performance comparison for simple RDF graph analytical queries

ProductType objects were grouped based on their prefixes. For example, type triples with object ProductType1, ProductType10, ProductType111, and so on, were grouped into the same equivalence class. The pre-processing phase for *BSBM-500K* took ∼22 minutes on a 10-node cluster to compute all the subject-triple groups in the dataset. The pre-processing phase for *BSBM-2M* took ∼18 minutes on a 60-node cluster and produced 136,364,658 subject triplegroups (66 equivalence class files with a total size of ∼60GB after prefix substitution). The pre-processing phase for the Chem2Bio2RDF dataset took ∼22mins on a 50-node cluster and produced 54,164,628 subject triplegroups (370 equivalence class files with a total size of ∼24GB after prefix substitution). The pre-processing phase for PubMed dataset took ∼36mins on a 60-node cluster.

5.5.2 Evaluation of RDF analytical queries with single grouping-aggregation

In order to study the behavior of *RAPIDAnalytics* and *Hive* with varying selectivity of the associated graph pattern and the granularity of the groupings, four queries with a single grouping-aggregation phase (single graph pattern) were evaluated. Queries $G1$ and $G2$ compute aggregations over all product offers of a particular type, i.e., `GROUP BY` ALL, while queries $G3$ and $G4$ compute aggregations over product offers of a particular type over groupings of product feature, i.e., `GROUP BY` feature. Queries $G1$ and $G3$ pertain to products of type ProductType1 (low selectivity), queries $G2$ and $G4$ pertain to products of type ProductType9 (high selectivity).

Figure 5.9(a) shows a performance comparison of *Hive* and *RAPIDAnalytics* for *BSBM-500K* on a 10-node Hadoop cluster. All four queries contain a single graph pattern with two star subpatterns. Hence, for each query, *Hive* required a total of 4 MR cycles – MR_1 and MR_2 to compute the two star patterns, MR_3 to join the two stars, and MR_4 to compute the grouping-aggregation phase. In cases where the (n-1) of the n joining relations are small enough to fit in memory, Hive converts the join into a map-join, and executes it as a map-only MR cycle.

This was observed in all four queries, while computing the star pattern involving small relations of products of type ProductType1 and ProductType9. In general, high-selectivity queries $G2$ and $G4$ produce less intermediate results during the graph pattern processing phase, resulting in less scan costs, I/O, and network transfers overall. Also, in general, the grouping-aggregation cycles in Hive are very light-weight due to several optimizations such as push down of `PROJECT`s based on the final aggregations. Additionally, Hive enables partial aggregation during the last join (cycle MR_3), which reduces the amount of intermediate data written into HDFS at the end of MR_3 and also reduces the input scan costs for cycle MR_4. The benefit of such an optimization is seen largely in the case of queries $G1$ and $G2$ which Group by ALL, thereby allowing a more aggressive partial aggregation of tuples during the join phase. On the other hand, *RAPIDAnalytics* executes all the four queries in 2 MR cycles – MR_1 for graph pattern processing (group-filtering based on star patterns and join between the two stars), and MR_2 for the *MD-Join* operation. *RAPIDAnalytics* shows a consistent performance gain of ∼80% over Hive for all four queries.

Discussion. The performance gain of *RAPIDAnalytics* can be attributed to a combination of factors. First, unlike the VP-based approach which processes irrelevant triples that may not eventually join, *RAPIDAnalytics* only loads subject triplegroups (groups of triples that join on the subject) from relevant equivalence classes. Second, the star pattern computation phase, which requires an MR cycle in Hive, is evaluated as a simple group-filtering on the set of subject triplegroups. Third, though *RAPIDAnalytics* does not currently use any compression scheme like Hive's ORC file format, *RAPIDAnalytics* benefits from the compact representation of intermediate results with multi-valued properties such as $productFeature$. Also, though *RAPIDAnalytics* does not currently enable partial aggregation during the preceding join phase, the hash-based aggregation during the map phase of `TG_MDJ` reduces the intermediate data and network transfers during the grouping-aggregation phase.

5.5.3 Evaluation of analytical queries with multiple grouping-aggregations

Four queries $MG1$, $MG2$, $MG3$, and $MG4$, all associated with two graph patterns and two grouping-aggregation phases, were evaluated. Queries $MG1$ and $MG3$ have low selectivity, while queries $MG2$ and $MG4$ have higher selectivity. Figure 5.10 shows a performance comparison of *Hive (Naive)*, *Hive (MQO)*, *RAPID+ (Naive)*, and *RAPIDAnalytics* for the two datasets. Queries $MG1$ and $MG2$ contain graph patterns with two star subpatterns, thus requiring 3 MR cycles for computation of each graph pattern in naive Hive (a total of 6 MR cycles for graph pattern processing). This is followed by 2 MR cycles for the two grouping-aggregation phases, and an additional cycle to join the aggregated results (a total of 9 MR cycles). On the other hand, the MQO-based Hive approach executes the composite graph pattern in just

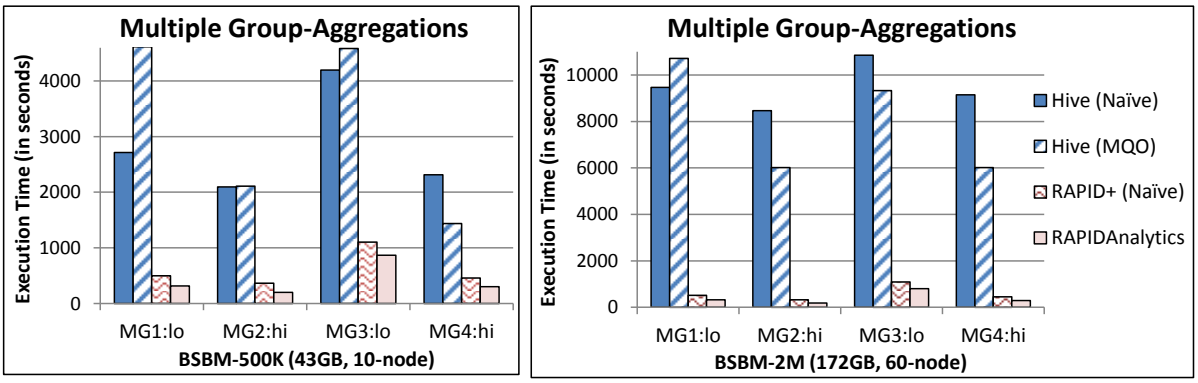


Figure 5.10: A performance comparison for multi-grouping RDF graph analytical queries

3 MR cycles. This is followed by 4 MR cycles to extract the distinct combinations matching the original patterns, the two grouping-aggregation phases, and an additional cycle to join the aggregated results (a total of 7 MR cycles). The naive RAPID+ requires 2 MR cycles for each subquery (1 MR cycle for graph pattern matching, 1 MR cycle for grouping-aggregation) and a map-only cycle to join the aggregated results, thus requiring a total of 5 MR cycles. On the other hand, *RAPIDAnalytics* $MG1$ and $MG2$ in 3 MR cycles – MR_1 to compute the composite graph pattern using `TG_AlphaJoin`, MR_2 for parallel evaluation of the two grouping-aggregations using `TG_MDJ`, and a map-only cycle MR_3 to join the aggregated tripegroups.

Queries MG_3 and MG_4 involve two graph patterns, each consisting of three star patterns (similar to the running example in this chapter). The sequential graph pattern processing phase in naive Hive results in a total of 8 MR cycles, followed by 2 MR cycles for grouping-aggregation phases, and a final cycle to join the aggregated results (a total of 11 MR cycles). MQO-based Hive approach takes half the number of cycles for the evaluation of the composite graph pattern, resulting in a total of 8 MR cycles. Naive RAPID+ requires 2 MR cycles for each graph pattern and 1 MR cycle for the associated grouping-aggregation (6 MR cycles for two subqueries), and a map-only cycle to join the associated join results (a total of 7 MR cycles). On the contrary, *RAPIDAnalytics* evaluates $MG3$ and $MG4$ in 4 MR cycles – MR_1 and MR_2 to compute the composite graph pattern using `TG_AlphaJoin`, MR_3 for parallel evaluation of the two grouping-aggregations using `TG_MDJ`, and a map-only cycle MR_4 to join the aggregated tripegroups.

Discussion. Though the number of MR cycles is less in *Hive(MQO)* when compared to naive Hive, the performance is worse than sequential execution of the original pattern subqueries in some cases. This is because Hive does not support materialized views or views with complex join expressions, and hence, the joined results matching the composite graph pattern in *Hive(MQO)* are stored in a regular Hive table. As a consequence of this, the HiveQL query that evaluates the composite graph pattern is not a subquery of the second HiveQL query that computes the

groupings and aggregations, and hence, (i) Hive cannot push down PROJECTs based on the final aggregation column, and (ii) Hive cannot partially aggregate tuples during the final join of the graph pattern processing phase. The effects of the preceding factors is likely to cause a higher overhead in low-selectivity queries such as $MG1$ and $MG3$ (refer Figure 5.10(a)). In general, *RAPIDAnalytics*'s algebraic optimization to group and aggregate on a composite graph pattern showed 30-45% gains over sequential evaluation of graph patterns and grouping-aggregations using naive RAPID+.

Scalability Study. Figure 5.9(b) and Figure 5.10(b) show performance comparisons of the 8 queries on a larger dataset *BSBM-2M*, evaluated on a 60-node Hadoop cluster. One observation is that though we increased the size of the cluster, not all nodes were utilized for all MR cycles. Hive's ORC File format compresses the Hive input data as well as intermediate results, which results in initialization of less number of mappers when compared to uncompressed data. The less number of mappers seem to incur an overhead of decompression as well, which was seen more prominently in MR cycles with large number of input triples. On the other hand, the RAPID+ based approaches initiate more number of mappers for most MR cycles during the graph pattern processing phase, leading to better utilization of resources. For simple analytical queries, both *Hive* and *RAPIDAnalytics* showed similar trends on the small and large datasets and cluster sizes. For multi-grouping queries, the MQO-based Hive approach did better than Hive for most cases with the larger dataset. This could be attributed to higher savings in materialization of intermediate results, associated I/Os, and network transfers. *RAPIDAnalytics* showed 90-93% performance gains over *Hive (MQO)* for queries $MG1$ and $MG2$ on *BSBM-500K*, which further increased to 97% with *BSBM-2M*. Similar increase in performance was seen for queries $MG3$ and $MG4$, where performance gains of *RAPIDAnalytics* over *Hive (MQO)* increased from 78-81% to 93% with the larger dataset and larger cluster size.

5.5.4 Evaluation of real-world RDF analytical queries

Evaluation on the Chem2Bio2RDF queries included five queries $G5$-$G9$ with single grouping-aggregation, and five queries $MG6 - MG10$ with multiple grouping-aggregation phases. The evaluated queries have varying number of star sub patterns in the associated graph patterns as represented in Table 5.2. Further, the density of star patterns (number of triple patterns per star pattern) also varies across queries, thus varying the number of join operations required for the graph pattern matching phase. Figure 5.11(a) shows a performance comparison of *Hive* and *RAPIDAnalytics* for queries $G5$-$G9$ with single grouping-aggregation. Query $G5$ involves 6 join operations (including star-joins) requiring 6 MR cycles, following an MR cycle for grouping-aggregation. However, due to the small size of the joining VP tables (attributed to small number of triples as well as aggressive compression), Hive is able to evaluate all 6 join operations as

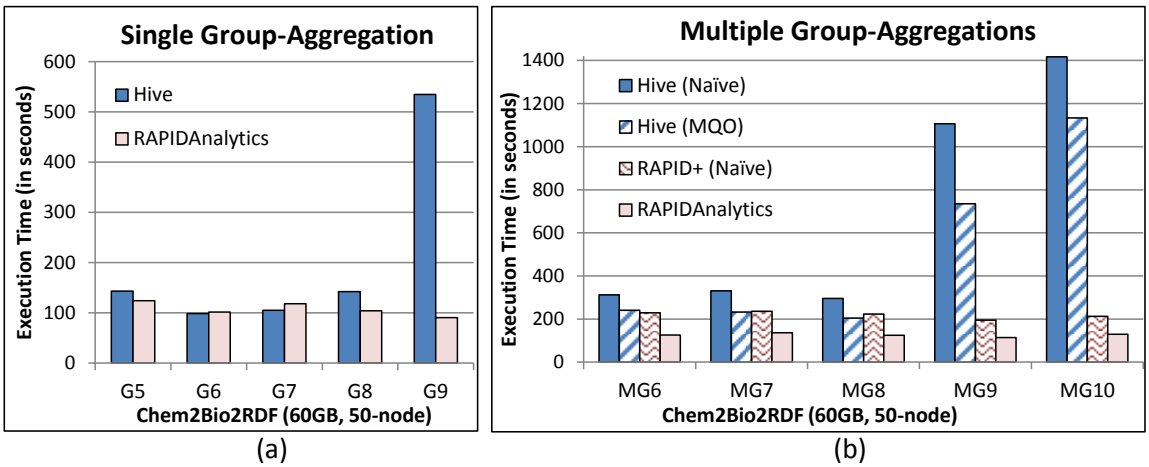


Figure 5.11: Evaluating real-world RDF analytical queries (Chem2Bio2RDF, A Chemogenomics RDF Data Warehouse)

a map-only join, with savings in I/O and network transfer costs. Further, the final join and the grouping-aggregation phase is executed in the same MR cycle, thus requiring only 6 MR cycles, 5 of which are map-only cycles. On the contrary, *RAPIDAnalytics* does not currently support map-only `TG_Join` between triplegroups. Hence, a map-reduce cycle is initiated even if the size of input data is small. *RAPIDAnalytics* evaluates query $G5$ in 5 full MR cycles, MR_1 to MR_4 for the graph pattern matching phase and MR_5 for grouping-aggregation. Query $G6$ with three star patterns (requiring 4 join operations) is evaluated in 4 MR cycles in Hive – 3 map-only cycles for first 3 joins, and last map-reduce cycle that computes the 4th join in the map phase, followed by grouping-aggregation in the reduce. *RAPIDAnalytics* initiates 2 MR cycles for the graph pattern matching phase, followed by a third cycle for the grouping-aggregation. Similarly, query $G7$ and $G8$ also involve properties with less number of triples (small VP tables), and hence Hive applies optimizations such as map-only joins and coalescing of join and grouping-aggregation phases. The benefit of such optimizations is clearly seen in the case of $G7$, where *RAPIDAnalytics* takes 12 additional seconds when compared to Hive. The last query $G9$ involves medline properties with larger sized VP tables. Thus, Hive evaluates the corresponding first star pattern in MR_1 as a full map-reduce cycle. The second join and the grouping-aggregation phases are evaluated in MR_2. *RAPIDAnalytics* also evaluates $G9$ using 2 MR cycles, but shows 83% performance gain over Hive. In addition to computing join graph pattern matching in a single MR cycle, *RAPIDAnalytics* initiates more number of mappers and reducers compared to Hive.

Figure 5.11(b) shows the evaluation results for queries with multiple grouping-aggregation phases. In order to study the benefit of Hive's map-join optimizations, the evaluation considered a mix of queries – $MG6$-$MG8$ involve properties with high selectivity (small VP relations), and

queries $MG9$-$MG10$ involve medline properties with slightly large VP relations. As expected, the Hive-based approaches evaluate star patterns with small VP relations as map-only joins. For example, naive Hive evaluates query $MG6$ using 13 MR cycles, 11 of which are map-only cycles. For the same query, the MQO-based Hive approach requires 8 MR cycles, 6 of which are map-only cycles. Naive RAPID+ evaluates $MG6$ using 7 MR cycles (all map-reduce cycles), with execution times almost comparable with *Hive (MQO)*. On the contrary, *RAPIDAnalytics* requires a total of 4 MR cycles (2MR cycles for evaluating the composite graph pattern, 1 MR cycle for parallel grouping-aggregation, and 1 map-only cycle to join the aggregated triple-groups). In general, even though though the Hive-based approaches evaluate most of the joins in $MG6 - MG8$ as map-joins (as seen in simple analytical queries), *RAPIDAnalytics* shows a performance gain of 40-50% over *Hive (MQO)* and 60% gains over naive Hive for queries $MG6$-$MG8$. In the case of queries $MG9$-$MG10$, the findings are similar to the BSBM dataset, with *RAPIDAnalytics* showing close to 90% performance gain over the Hive approaches.

Table 5.4: Evaluation of real-world queries on PubMed dataset (execution time in seconds)

Query	**PubMed** (230GB dataset, 60-node cluster)			
	Hive (Naive)	**Hive (MQO)**	**RAPID+ (Naive)**	**RAPID Analytics**
MG11	2111	1753	229	124
MG12	2771	2898	229	126
MG13	> 120min*	15060	1102	651
MG14	18713	9124	756	462
MG15	13746	7320	619	338
MG16	10777	5795	464	237
MG17	2210	1851	226	118
MG18	5654	4817	306	202
* Eventually failed due to insufficient HDFS disk space.				

Results for the *Pubmed* dataset are summarized in Table 5.4. Queries $MG11$-$MG18$ are all analytical queries with two grouping-aggregation constraints with varying granularity of required groupings. The Hive approaches evaluated all star patterns as map-reduce joins due to the sizes of the query-relevant VP relations. Queries $MG11$-$MG12$ and $MG17$-$MG18$ compute groupings over PubMed records, the associated grants, and the countries where the grants are issued. Queries $MG13$-$MG16$ compute groupings based on publication type and authors of PubMed records and aggregate the number of Medical Subject (MeSH) Headings (query $MG13$) or associated chemicals (queries $MG14$-$MG16$). Further, selectivity of the queries were varied by querying different types of publications, e.g., $MG15$ and $MG16$ have similar query

structure except that $MG15$ retrieves PubMed records with publication type "Journal Article" while $MG16$ concerns publications of type "News" (higher selectivity when compared to records of type journal article). Across all queries, *RAPIDAnalytics* showed improvements of above 93% over both Hive approaches. Hive performed the worst for queries $MG13$-$MG16$ that involve large VP relations (MeSH heading and chemical), mainly due to the initiation of less number of mappers. For example, the VP relation for MeSH heading with ∼184 million triples is compressed using Hive's ORC format into a file with size ∼640MB. While evaluating a star pattern that involves property MeSH heading (query $MG13$), Hive initiates only 6 mappers (based on the compressed file sizes), thereby burdering the mappers with the decompression and map-side processing for a large number of triples. As a result, each star-join cycle involving the property MeSH heading consumes about 3 hours of evaluation time in both Hive approaches. Furthermore, while the Hive MQO approach eventually finished execution for query $MG13$, the naive Hive approach failed while computing the second graph pattern due to insufficient disk space. This is because one of the star-join cycles produces join output of size ∼190GB, which is materialized twice in the case of sequential execution of graph patterns, thus increasing the overall demand of required HDFS disk space. On the contrary, *RAPIDAnalytics* benefits from the concise representation of intermediate results using the NTGA approach while representing join results involving the multi-valued property MeSH heading. Further, the shared execution of graph patterns in *RAPIDAnalytics*, results in less number of materialization steps and less demand on required disk space. Overall, *RAPIDAnalytics* resulted in 40-48% performance gains over the sequential execution of graph patterns in *RAPID+*.

Discussion. The vertical-partitioning (VP) approach coupled with the ORC file format can be beneficial for RDF analytical queries that involve properties with high selectivity. This is because joins involving small sized VP relations can be evaluated as a map-only join in Hive, thus reducing the map-side local disk I/O and the network transfer costs. Similar approaches can be integrated in RAPID+ to allow map-side joins between triplegroups. Current implementation supports map-side joins only in `TG_AggJoin`, which can be extended to `TG_AlphaJoin` as well. The performance improvement of *RDFAnalytics* over naive RAPID+ could be seen in all queries. In general, the algebraic optimization techniques in *RDFAnalytics* that enable shared computations in the graph pattern matching phase as well as the grouping-aggregation phases, are beneficial for all multi-group queries, irrespective of the selectivity of the involved properties.

5.6 Chapter Summary

In this chapter, we presented an algebraic optimization technique for supporting efficient and scalable processing of analytical queries on RDF data models. We presented a refactoring of

RDF analytical queries based on a new set of logical operators, that achieves shared execution of common subexpressions. Such a refactoring enables parallel evaluation of groupings as well as aggregations, leading to reduced scans, I/Os, and processing costs, particularly beneficial for distributed processing on Cloud platforms. Experiments on real-world and synthetic benchmarks confirmed that such a rewriting can achieve up to 10X speedup over relational-style SPARQL query plans executed on popular Cloud systems.

Chapter 6

Conclusion and Future Work

6.1 Overview of Dissertation

This dissertation studied the problem of supporting efficient evaluation of RDF analytical queries on MapReduce-based parallel data processing systems. In Chapter 1 , we present a motivational example to highlight the challenges in relational-style processing of multi-aggregation RDF analytical queries on MapReduce. We identify two main optimization goals for RDF analytics on MapReduce – (1) minimize the length of MapReduce execution workflow, and (2) efficient management of intermediate results. We also identify sharing opportunities across the graph pattern matching phase and the grouping-aggregation phases, that would help us achieve these goals. In Chapter 2, we present an algebraic optimization approach based on an alternative data model and algebra called the *Nested TripleGroup Data Model and Algebra*(NTGA), that leads to efficient MapReduce execution plans with reduced numbers of MR cycles. We also establish a '*content equivalence*' between expressions in NTGA and relational algebra expressions representing a class of graph pattern queries. In Chapter 3, we discuss strategies for efficient management of intermediate results while evaluating graph pattern queries that involve multi-valued properties. We show how nesting-aware operators can be used with NTGA's nested data model to mitigate the effects of redundancy in intermediate results while evaluating such queries. We introduce logical and physical operators that enable *partial* and *lazy* unnesting strategies for joins on multi-valued properties. In Chapter 4, we discuss the issues with flexible querying of semantically integrated RDF data warehouses using graph pattern queries with unbound properties ('don't care edges'). We build on the advantages of NTGA's nested model and extend the suite of NTGA operators to handle unbound-property queries. We show how a non-relational interpretation of unbound-property queries using NTGA can enable concise representation of intermediate results, which is critical in reducing the I/O footprint of MapReduce execution workflows. In Chapter 5, we discuss algebraic optimization techniques

to evaluate RDF analytical queries with multiple grouping-aggregation phases. At the end of each chapter, we present evaluation results on synthetic benchmark and real-world datasets. Evaluation results confirm the superiority of NTGA-based query plans in achieving shortened execution workflows and reducing the I/O footprint while evaluating RDF analytical queries on MapReduce.

6.2 Future Directions

In the future, we would like to further study the problem of multi-query optimization of RDF analytical queries. Specifically, we would like to investigate the problem of extending NTGA-based approach to share scans, computations, and intermediate data references across simple pattern matching queries and RDF analytical queries. To further bring down the execution times for RDF analytical query processing, we would like to investigate physical storage models that allow more optimized execution plans. We would also like to investigate the problem of extending our approach for more dynamic BI workloads that involve analytics over RDF data streams.

REFERENCES

[1] Apache pig. http://wiki.apache.org/pig.

[2] Berlin SPARQL Benchmark (BSBM) - Business Intelligence Use Case 3.1. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BusinessIntelligenceUseCase/.

[3] Billion triple challenge. http://challenge.semanticweb.org/.

[4] Bio2RDF Demo Queries. http://sourceforge.net/apps/mediawiki/bio2rdf/index.php?title=Demo_queries.

[5] ORC File Format. http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds_Hive/orcfile.html.

[6] D.J. Abadi, A. Marcus, S.R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[7] S. Abiteboul, P.C. Fischer, and H.J. Schek. *Nested relations and complex objects in databases*, volume 361. springer, 1989.

[8] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[9] Azza Abouzied, Kamil Bajda-Pawlikowski, Jiewen Huang, Daniel J Abadi, and Avi Silberschatz. Hadoopdb in action: building real world applications. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1111–1114. ACM, 2010.

[10] F.N. Afrati and J.D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.

[11] Michael O Akinde and Michael H Böhlen. Generalized md-joins: Evaluation and reduction to sql. In *Databases in Telecommunications II*, pages 52–67. Springer, 2001.

[12] Michael O Akinde, Michael H Böhlen, Theodore Johnson, Laks VS Lakshmanan, and Divesh Srivastava. Efficient olap query processing in distributed data warehouses. *Information Systems*, 28(1):111–135, 2003.

[13] Jens Albrecht and Wolfgang Lehner. On-line analytical processing in distributed data warehouses. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 78–85. IEEE, 1998.

[14] Kemafor Anyanwu, HyeongSik Kim, and Padmashree Ravindra. Algebraic optimization for processing graph pattern queries in the cloud. *Internet Computing, IEEE*, 17(2):52–61, 2013.

[15] Kemafor Anyanwu, Padmashree Ravindra, and HyeongSik Kim. Algebraic optimization of rdf graph pattern queries on mapreduce. In *Large Scale and Big Data - Processing and Management.*, pages 183–228. 2014.

[16] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. *The Semantic Web*, pages 722–735, 2007.

[17] Amos Bairoch, Lydie Bougueleret, Severine Altairac, Valeria Amendolia, Andrea Auchincloss, Ghislaine Argoud Puy, Kristian Axelsen, Delphine Baratin, Marie-Claude Blatter, Brigitte Boeckmann, et al. The universal protein resource (uniprot). *Nucleic Acids Research*, 36:D190–D195, 2008.

[18] François Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. Bio2rdf: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, 41(5):706–716, 2008.

[19] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

[20] A. Bialecki, M. Cafarella, D. Cutting, and O. O'MALLEY. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene.apache.org/hadoop*, 11, 2005.

[21] C. Bizer, A. Jentzsch, and R. Cyganiak. State of the lod cloud. *Online: http://www4.wiwiss.fu-berlin. de/lodcloud/state*, 2011.

[22] C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[23] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 975–986. ACM, 2010.

[24] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[25] Don Chamberlin. *Using the new DB2: IBM's object-relational database system*. Morgan Kaufmann Publishers Inc., 1996.

[26] D. Chatziantoniou, T. Johnson, M. Akinde, and S. Kim. The md-join: An operator for complex olap. In *Proceedings. 17th International Conference on Data Engineering (ICDE)*, pages 524–533. IEEE, 2001.

[27] Damianos Chatziantoniou. Ad hoc olap: Expression and evaluation. In *ICDE*, page 250, 1999.

[28] Damianos Chatziantoniou and Yannis Sotiropoulos. Asset queries: A set-oriented and column-wise approach to modern olap. In *Enabling Real-Time Business Intelligence*, pages 66–83. Springer, 2010.

[29] Damianos Chatziantoniou and Elias Tzortzakakis. Asset queries: a declarative alternative to mapreduce. *ACM SIGMOD Record*, 38(2):35–41, 2009.

[30] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[31] Bin Chen, Xiao Dong, Dazhi Jiao, Huijun Wang, Qian Zhu, Ying Ding, and David J Wild. Chem2bio2rdf: a semantic framework for linking and data mining chemogenomic and systems chemical biology data. *BMC bioinformatics*, 11(1):255, 2010.

[32] Huajun Chen, Tong Yu, and Jake Y Chen. Semantic web meets integrative biology: a survey. *Briefings in bioinformatics*, 14(1):109–125, 2013.

[33] Lei Chen, Christopher Olston, and Raghu Ramakrishnan. Parallel evaluation of composite aggregate queries. In *24th IEEE International Conference on Data Engineering*, pages 218–227. IEEE, 2008.

[34] Kei-Hoi Cheung, Kevin Y Yip, Andrew Smith, Andy Masiar, Mark Gerstein, et al. Yeasthub: a semantic web use case for integrating data in the life sciences domain. *Bioinformatics*, 21(suppl 1):i85–i96, 2005.

[35] Dario Colazzo, François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. RDF Analytics: Lenses over Semantic Graphs. In *Proceedings of the 23rd international conference on World wide web (WWW)*, pages 467–478, 2014.

[36] R Cyganiak, D Reynolds, and J Tennison. The rdf data cube vocabulary, w3c working draft 05 april 2012. *World Wide Web Consortium*, 2012.

[37] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[38] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. *Proceedings of the VLDB Endowment*, 5(6):586–597, 2012.

[39] Lorena Etcheverry and Alejandro A Vaisman. Enhancing olap analysis with web cubes. In *The Semantic Web: Research and Applications*, pages 469–483. Springer, 2012.

[40] Goetz Graefe, Usama M Fayyad, Surajit Chaudhuri, et al. On the efficient gathering of sufficient statistics for classification from large sql databases. In *KDD*, pages 204–208, 1998.

[41] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 152–159, 1996.

[42] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. Implementing data cubes efficiently. In *ACM SIGMOD Record*, volume 25, pages 205–216. ACM, 1996.

[43] S. Harris and A. Seaborne. Sparql 1.1 query language. *W3C Working Draft*, 14, 2010.

[44] S. Helmer, G. Moerkotte, et al. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 386–395. Citeseer, 1997.

[45] J. Huang, D.J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[46] M. Husain, J. McGlothlin, M.M. Masud, L. Khan, and B.M. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(9):1312–1327, 2011.

[47] M.F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data intensive query processing for large rdf graphs using cloud computing tools. In *2010 IEEE 3rd International Conference on Cloud Computing (CLOUD*, pages 1–10. IEEE, 2010.

[48] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[49] Benedikt Kämpgen and Andreas Harth. No size fits all–running the star schema benchmark with sparql and rdf aggregate views. In *The Semantic Web: Semantics and Big Data*, pages 290–304. Springer, 2013.

[50] Benedikt Kämpgen, Sean ORiain, and Andreas Harth. Interacting with statistical linked data via olap operations. In *Proceedings of Interacting with Linked Data (ILD 2012)*, pages 36–49. Citeseer, 2012.

[51] H.S. Kim, P. Ravindra, and K. Anyanwu. From sparql to mapreduce: The journey using a nested triplegroup algebra. *Proceedings of the VLDB Endowment*, 4(12), 2011.

[52] H.S. Kim, P. Ravindra, and K. Anyanwu. Scan-sharing for optimizing rdf graph pattern matching on mapreduce. In *IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 139–146. IEEE, 2012.

[53] G. Klyne, J.J. Carroll, and B. McBride. Resource description framework (rdf): Concepts and abstract syntax. *W3C recommendation*, 10, 2004.

[54] Spyros Kotoulas, Jacopo Urbani, Peter Boncz, and Peter Mika. Robust Runtime Optimization and Skew-resistant Execution of Analytical SPARQL Queries on Pig. In *The Semantic Web–ISWC 2012*, pages 247–262. Springer, 2012.

[55] Hugo YK Lam, Luis Marenco, Tim Clark, Yong Gao, June Kinoshita, Gordon Shepherd, Perry Miller, Elizabeth Wu, Gwendolyn T Wong, Nian Liu, et al. Alzpharm: integration of neurodegeneration data using rdf. *BMC bioinformatics*, 8(Suppl 3):S4, 2007.

[56] R. Lawrence. Using slice join for efficient evaluation of multi-way joins. *Data & Knowledge Engineering*, 67(1):118–139, 2008.

[57] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 666–677, 2012.

[58] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *31st International Conference on Distributed Computing Systems (ICDCS)*, pages 25–36. IEEE, 2011.

[59] Chengkai Li, Bin He, Ning Yan, Muhammad Safiullah, and Rakesh Ramegowda. Set Predicates in SQL: Enabling Set-Level Comparisons for Dynamically Formed Groups. *IEEE Transactions on Knowledge and Data Engineering*, PP(99):1, 2012.

[60] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: a transformation-based optimizer for mapreduce workflows. *VLDB*, pages 1196–1207, 2012.

[61] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[62] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2003.

[63] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *IEEE 27th International Conference on Data Engineering (ICDE)*, pages 183–194. IEEE, 2011.

[64] Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, 2010.

[65] A. Newman, J. Hunter, Y.F. Li, C. Bouton, and M. Davis. A scale-out rdf molecule store for distributed processing of biomedical data. In *Semantic Web for Health Care and Life Sciences Workshop*, 2008.

[66] A. Newman, Y.F. Li, and J. Hunter. Scalable semantics–the silver lining of cloud computing. In *IEEE Fourth International Conference on eScience*, pages 111–118. IEEE, 2008.

[67] Marc-Alexandre Nolin, Peter Ansell, François Belleau, Kingsley Idehen, Philippe Rigault, Nicole Tourigny, Paul Roe, James M Hogan, and Michel Dumontier. Bio2rdf network of linked data. In *Semantic Web Challenge; ISWC 2008*, 2008.

[68] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *Proceedings of the VLDB Endowment*, 3(1-2):494–505, 2010.

[69] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V.B.N. Rao, V. Sankarasubramanian, S. Seth, et al. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1081–1090. ACM, 2011.

[70] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[71] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *ACM Sigmod Record*, volume 26, pages 38–49. ACM, 1997.

[72] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2rdf: adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 397–400. ACM, 2012.

[73] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277, 2005.

[74] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf, W3C Recommendation, 2008. *URL http://www.w3.org/TR/rdf-sparql-query*, 2010.

[75] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. Technical report, Technical Report YL-2012-002, Yahoo, 2012.

[76] P. Ravindra, V.V. Deshpande, and K. Anyanwu. Towards scalable rdf graph analytics on mapreduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud (MDAC)*, page 5. ACM, 2010.

[77] P. Ravindra, H.S. Kim, and K. Anyanwu. An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. *The Semantic Web: Research and Applications*, pages 46–61, 2011.

[78] P. Ravindra, H.S. Kim, and K. Anyanwu. To nest or not to nest, when and how much: representing intermediate results of graph pattern queries in mapreduce based processing. In *Proceedings of the 4th International Workshop on Semantic Web Information Management (SWIM)*, page 5. ACM, 2012.

[79] Padmashree Ravindra. Towards optimization of rdf analytical queries on mapreduce. In *IEEE 30th International Conference on Data Engineering Workshops (ICDEW)*, pages 335–339. IEEE, 2014.

[80] Padmashree Ravindra and Kemafor Anyanwu. Scalable processing of flexible graph pattern queries on the cloud. In *Proceedings of the 22nd international conference on World Wide Web (WWW) companion*, pages 169–170. International World Wide Web Conferences Steering Committee, 2013.

[81] Padmashree Ravindra and Kemafor Anyanwu. Nesting strategies for enabling nimble mapreduce dataflows for large rdf data. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(1):1–26, 2014.

[82] K. Rohloff and R.E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, page 4. ACM, 2010.

[83] H.E. Schaffer, S.F. Averitt, M.I. Hoit, A. Peeler, E.D. Sills, and M.A. Vouk. Ncsu's virtual computing lab: a cloud computing solution. *Computer*, 42(7):94–97, 2009.

[84] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel. An experimental comparison of rdf data management approaches in a sparql benchmark scenario. *The Semantic Web-ISWC 2008*, pages 82–97, 2008.

[85] Nigel Shadbolt, Wendy Hall, and Tim Berners-Lee. The semantic web revisited. *Intelligent Systems, IEEE*, 21(3):96–101, 2006.

[86] Ambuj Shatdal and Jeffrey F Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD Record*, volume 24, pages 104–114. ACM, 1995.

[87] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563, 2008.

[88] M. Sintek and M. Kiesel. Rdfbroker: A signature-based high-performance rdf store. *The Semantic Web: Research and Applications*, pages 363–377, 2006.

[89] R. Sridhar, P. Ravindra, and K. Anyanwu. Rapid: Enabling scalable ad-hoc analytics on the semantic web. *The Semantic Web-ISWC 2009*, pages 715–730, 2009.

[90] S. Stefanova and T. Risch. Optimizing unbound-property queries to rdf views of relational databases. In *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, page 43, 2011.

[91] S. Stefanova and T. Risch. Optimizing unbound-property queries to rdf views of relational databases. In *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, page 43, 2011.

[92] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: Graph algorithms for the (semantic) web. *The Semantic Web–ISWC 2010*, pages 764–780, 2010.

[93] Y. Tanimura, A. Matono, S. Lynden, and I. Kojima. Extensions to the pig data processing platform for scalable rdf data processing using hadoop. In *IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, pages 251–256. IEEE, 2010.

[94] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[95] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal. Owl reasoning with webpie: calculating the closure of 100 billion triples. *The Semantic Web: Research and Applications*, pages 213–227, 2010.

[96] J. Urbani, S. Kotoulas, E. Oren, and F. Van Harmelen. Scalable distributed reasoning using mapreduce. *The Semantic Web-ISWC 2009*, pages 634–649, 2009.

[97] Rares Vernica, Michael J Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM, 2010.

[98] M.E. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in sparql queries. *The Semantic Web: Research and Applications*, pages 228–242, 2010.

[99] X. Wang, C. Olston, A.D. Sarma, and R. Burns. Coscan: cooperative scan sharing in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 11. ACM, 2011.

[100] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[101] Ming-Chuan Wu and Alejandro P Buchmann. Encoded bitmap indexing for data warehouses. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, pages 220–230. IEEE, 1998.

[102] S. Wu, F. Li, S. Mehrotra, and B.C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.

[103] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14, 2008.

[104] Amrapali Zaveri, Ricardo Pietrobon, Soren Auer, Jens Lehmann, Michael Martin, and Timofey Ermilov. Redd-observatory: Using the web of data for evaluating the research-disease disparity. In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology-Volume 01*, pages 178–185. IEEE Computer Society, 2011.

APPENDICES

Appendix A

Evaluated Basic Graph Pattern Queries

A.1 Case Study: Grouping of Joins

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

SPARQL Query Q1a (Object-Subject Join between Stars).

```
SELECT * WHERE {
   ?vend foaf:homepage ?vHpage; bsbm:country ?vcountry; rdfs:label ?vLabel .
   ?offer bsbm:vendor ?vend; bsbm:price ?price; bsbm:validFrom ?vFrom . }
```

>> Hive script: SJ-per-cycle approach.

```
CREATE TABLE OV_Star(S1 String, S2 String, P1 String, O1 String, P2 String,
O2 String, P3 String, O3 String, P4 String, O4 String, P5 String, O5 String,
P6 String, O6 String);
INSERT OVERWRITE TABLE OV_Star
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2,
J1.p3, J1.o3, J2.p1, J2.o1, J2.p2, J2.o2, J2.p3, J2.o3 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1,
T2.predicate p2, T2.object o2, T3.predicate p3, T3.object o3
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject)
JOIN triples T3 ON (T1.subject=T3.subject) WHERE
```

```
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/country’
and T2.predicate=‘ <http://xmlns.com/foaf/0.1/homepage’ and
T3.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2 ON
(T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’ and
T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’ and
T3.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/validFrom’)
J2 ON (TRIM(J1.s1)=TRIM(J2.o1));
```

>> Hive script: Sel-SJ-First.

```
CREATE TABLE OV_Opt(S1 String, S2 String, P1 String, O1 String, P2 String,
O2 String, P3 String, O3 String, P4 String, O4 String, P5 String, O5 String,
P6 String, O6 String);
INSERT OVERWRITE TABLE OV_Opt
select J1.s1, J1.s2, J1.p1, J1.o1, J1.p2, J1.o2, J1.p3, J1.o3, J1.p4, J1.o4,
T11.predicate, T11.object, T12.predicate, T12.object FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3, T4.subject s2, T4.predicate p4, T4.object o4
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) JOIN triples T3
ON (T1.subject=T3.subject) JOIN triples T4 ON (T1.subject=TRIM(T4.object)) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/country’
and T2.predicate=‘ <http://xmlns.com/foaf/0.1/homepage’ and
T3.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’ and
T4.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’)J1
JOIN triples T11 ON (J1.s2=T11.subject)
JOIN triples T12 ON (J1.s2=T12.subject) WHERE
T11.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’ and
T12.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/validFrom’;
```

SPARQL Query Q1b (Q1a with additional filter).

```
SELECT * WHERE {
   ?vend foaf:homepage ?vHpage; ?vend rdfs:label ?vLabel;
         bsbm:country <http://downlode.org/rdf/iso-3166/countries#GB .
   ?offer bsbm:vendor ?vend; bsbm:price ?price; bsbm:validFrom ?vFrom . }
```

>> Hive script: SJ-per-cycle approach.

```
CREATE TABLE OV_Star(S1 String,S2 String,P1 String,O1 String,P2 String,O2 String,
P3 String,O3 String,P4 String,O4 String,P5 String,O5 String,P6 String,O6 String);
INSERT OVERWRITE TABLE OV_Star
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2, J1.p3, J1.o3, J2.p1, J2.o1,
J2.p2, J2.o2, J2.p3, J2.o3 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/country’
and T1.object=‘ <http://downlode.org/rdf/iso-3166/countries#GB’ and
T2.predicate=‘ <http://xmlns.com/foaf/0.1/homepage’ and
T3.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’ and
T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’ and
T3.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/validFrom’)
J2 ON (TRIM(J1.s1)=TRIM(J2.o1));
```

>> Hive script: Sel-SJ-First.

```
CREATE TABLE OV_Opt(S1 String,S2 String,P1 String,O1 String,P2 String,O2 String,
P3 String,O3 String,P4 String,O4 String,P5 String,O5 String,P6 String,O6 String);
INSERT OVERWRITE TABLE OV_Opt
select J1.s1, J1.s2, J1.p1, J1.o1, J1.p2, J1.o2, J1.p3, J1.o3, J1.p4, J1.o4,
T11.predicate, T11.object, T12.predicate, T12.object FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3, T4.subject s2, T4.predicate p4, T4.object o4
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) JOIN triples T3
ON (T1.subject=T3.subject) JOIN triples T4 ON (T1.subject=TRIM(T4.object)) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/country’
and T1.object=‘ <http://downlode.org/rdf/iso-3166/countries#GB’ and
T2.predicate=‘ <http://xmlns.com/foaf/0.1/homepage’ and
T3.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’ and T4.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’) J1 JOIN
triples T11 ON (J1.s2=T11.subject) JOIN triples T12 ON (J1.s2=T12.subject) WHERE
```

```
T11.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’ and
T12.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/validFrom’;
```

SPARQL Query Q2a (Object-Subject Join between Stars)

```
SELECT * WHERE {
   ?offer bsbm:vendor ?vend; bsbm:product ?prod .
   ?prod bsbm:productPropertyNumeric1 ?num1; rdfs:label ?prodLabel;
         bsbm:productFeature ?pf . }
```

>> Hive script: SJ-per-cycle approach.

```
CREATE TABLE OP_Star(S1 String,S2 String,P1 String,O1 String,P2 String,
O2 String, P3 String,O3 String,P4 String,O4 String,P5 String,O5 String);
INSERT OVERWRITE TABLE OP_Star
select J1.s1,J2.s1,J1.p1,J1.o1,J1.p3,J1.o3,J2.p1,J2.o1,J2.p2,J2.o2,J2.p3,J2.o3 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T3.predicate p3, T3.object o3
FROM triples T1 JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’)
J1 JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T2.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’
and T3.predicate=
‘ <http://www.w3.org/2000/01/rdf-schema#label’)J2 ON (TRIM(J1.o3)=TRIM(J2.s1));
```

>> Hive script: Sel-SJ-First.

```
CREATE TABLE OP_Opt(S1 String,S2 String,P1 String,O1 String,P2 String,O2 String,
P3 String,O3 String,P4 String,O4 String,P5 String,O5 String);
INSERT OVERWRITE TABLE OP_Opt
select J1.s1, T11.subject, J1.p1, J1.o1, J1.p2, J1.o2, T11.predicate, T11.object,
T12.predicate, T12.object, T13.predicate, T13.object FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) WHERE
```

```
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’ and
T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’)
J1 JOIN triples T11 ON (TRIM(J1.o2)=T11.subject) JOIN triples T12 ON
(TRIM(J1.o2)=T12.subject) JOIN triples T13 ON (TRIM(J1.o2)=T13.subject)
WHERE T11.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T12.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’
and T13.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’;
```

SPARQL Query Q2b (Q2a with additional filter).

```
SELECT * WHERE {
   ?offer bsbm:product ?prod; bsbm:vendor
   <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Vendor1;
   ?prod bsbm:productPropertyNumeric1 ?num1; rdfs:label ?prodLabel;
         bsbm:productFeature ?pf . }
```

>> Hive script: SJ-per-cycle approach.

```
CREATE TABLE OP_Star(S1 String,S2 String,P1 String,O1 String,P2 String,O2 String,
P3 String,O3 String,P4 String,O4 String,P5 String,O5 String);
INSERT OVERWRITE TABLE OP_Star
select J1.s1,J2.s1,J1.p1,J1.o1,J1.p3,J1.o3,J2.p1,J2.o1,J2.p2,J2.o2,J2.p3,J2.o3 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T3.predicate p3, T3.object o3
FROM triples T1 JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’
and T1.object=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Vendor1’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’)J1
JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3 FROM triples T1
JOIN triples T2 ON (T1.subject=T2.subject)
JOIN triples T3 ON (T1.subject=T3.subject) WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T2.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’
and T3.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’)J2
```

```
ON (TRIM(J1.o3)=TRIM(J2.s1));
```

>> Hive script: Sel-SJ-First.

```
CREATE TABLE OP_Opt(S1 String,S2 String,P1 String,O1 String,P2 String,O2 String,
P3 String,O3 String,P4 String,O4 String,P5 String,O5 String);
INSERT OVERWRITE TABLE OP_Opt
select J1.s1, T11.subject, J1.p1, J1.o1, J1.p2, J1.o2, T11.predicate, T11.object,
T12.predicate, T12.object, T13.predicate, T13.object FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’
and T1.object=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Vendor1’
and T2.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’)J1
JOIN triples T11 ON (TRIM(J1.o2)=T11.subject) JOIN triples T12 ON (TRIM(J1.o2)=
T12.subject) JOIN triples T13 ON (TRIM(J1.o2)=T13.subject) WHERE T11.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T12.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’
and T13.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’;
```

SPARQL Query Q3a (Object-Object Join between Stars).

```
SELECT * WHERE {
   ?review bsbm:reviewFor ?prod; rev:reviewer ?r; bsbm:rating1 ?rat1 .
   ?offer bsbm:product ?prod; bsbm:price ?price; bsbm:vendor ?vend . }
```

>> Hive script: SJ-per-cycle approach.

```
CREATE TABLE OR_Star(S1 String,S2 String,P1 String,O1 String,P2 String,O2 String,
P3 String,O3 String,P4 String,O4 String,P5 String,O5 String,P6 String,O6 String);
INSERT OVERWRITE TABLE OR_Star select J1.s1,J2.s1,J1.p1,J1.o1,J1.p2,J1.o2,
J1.p3,J1.o3,J2.p1,J2.o1,J2.p2,J2.o2,J2.p3,J2.o3
FROM (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2 ON
(T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/reviewFor’
and T2.predicate=‘ <http://purl.org/stuff/rev#reviewer’ and
```

```
T3.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/rating1’)
J1 JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’
and T2.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’ and
T3.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’)
J2 ON (TRIM(J1.o1)=TRIM(J2.o1));
```

>> Hive script: Sel-SJ-First.

```
CREATE TABLE OR_Opt(S1 String,S2 String,P1 String,O1 String,P2 String,O2 String,
P3 String,O3 String,P4 String,O4 String,P5 String,O5 String,P6 String,O6 String);
INSERT OVERWRITE TABLE OR_Opt
SELECT J1.s1, TJ.subject, J1.p1, J1.o1, J1.p2, J1.o2, J1.p3, J1.o3, TJ.predicate,
TJ.object, T11.predicate, T11.object, T12.predicate, T12.object FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2 ON
(T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/reviewFor’
and T2.predicate=‘ <http://purl.org/stuff/rev#reviewer’ and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/rating1’)J1
JOIN triples TJ ON (TRIM(J1.o1)=TRIM(TJ.object) and
TJ.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’)
JOIN triples T11 ON (TJ.subject=T11.subject) JOIN triples T12 ON
(TJ.subject=T12.subject) where T11.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’ and
T12.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’;
```

SPARQL Query Q3b (Q3a with additional filter).

```
SELECT * WHERE {
   ?review rev:reviewer bsbm-instances:dataFromRatingSite1/Reviewer1>;
           bsbm:reviewFor ?prod; bsbm:rating1 ?rat1 .
   ?offer bsbm:vendor ?vend; bsbm:price ?price; bsbm:product ?prod . }
```

>> Hive script: SJ-per-cycle approach.

```
CREATE TABLE OR_Star_Rev1(S1 String,S2 String,P1 String,O1 String,
```

```
P2 String, O2 String, P3 String,O3 String,P4 String,O4 String,
P5 String, O5 String,P6 String,O6 String);
INSERT OVERWRITE TABLE OR_Star select J1.s1,J2.s1,J1.p1,J1.o1,J1.p2,J1.o2,
J1.p3,J1.o3,J2.p1,J2.o1,J2.p2,J2.o2,J2.p3,J2.o3 FROM (
SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/reviewFor’
and T2.predicate=‘ <http://purl.org/stuff/rev#reviewer’ and T2.object =
‘ http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromRatingSite1/
Reviewer1’ and T3.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
vocabulary/rating1’)J1
JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’
and T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’)J2
ON (TRIM(J1.o1)=TRIM(J2.o1));
```

>> **Hive script: Sel-SJ-First.**

```
CREATE TABLE OR_Opt(S1 String,S2 String,P1 String,O1 String,P2 String,
O2 String, P3 String,O3 String,P4 String,O4 String,
P5 String,O5 String,P6 String,O6 String);
INSERT OVERWRITE TABLE OR_Opt SELECT J1.s1, TJ.subject, J1.p1, J1.o1, J1.p2,
J1.o2, J1.p3, J1.o3, TJ.predicate, TJ.object, T11.predicate, T11.object,
T12.predicate, T12.object FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject)
WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/reviewFor’
and T2.predicate=‘ <http://purl.org/stuff/rev#reviewer’ and T2.object=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromRatingSite1/
Reviewer1’ and T3.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
vocabulary/rating1’)J1 JOIN triples TJ ON (TRIM(J1.o1)=TRIM(TJ.object) and
```

```
TJ.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’)
JOIN triples T11 ON (TJ.subject=T11.subject) JOIN triples T12
ON (TJ.subject=T12.subject) where T11.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’
and T12.predicate
=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’;
```

Appendix B

Evaluated Graph Pattern Queries with Multi-valued Properties

B.1 Queries on Berlin SPARQL Benchmark Dataset

Two groups of queries involving multi-valued properties were formulated. The first group evaluates graph patterns where the multi-valued property is part of the result but the join value is single-valued (*non-MV join*). The second group consists of graph pattern queries where the join column is multi-valued i.e. the join is on the object of the multi-valued property (*MVJoin*).

```
   PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

SPARQL Query Low-1Star: 1 star pattern, low multiplicity MVP (rdf:type ∼6)

```
select * where {
   ?prod rdf:type ?pf; rdfs:label ?prodLabel; bsbm:productPropertyNumeric1 ?num1;
         bsbm:productPropertyNumeric2 ?num2 . }
```

>> **Hive script.**

```
CREATE TABLE Result_low1(Sub1 String, Prop1 String, Obj1 String, Prop2 String,
Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String);
INSERT OVERWRITE TABLE Result_low1 SELECT T1.subject sub1, T1.predicate prop1,
T1.object obj1, T2.predicate prop2, T2.object obj2, T3.predicate prop3,
T3.object obj3, T4.predicate prop4, T4.object obj4 FROM triples T1
```

```
JOIN triples T2 ON (T1.subject = T2.subject)
JOIN triples T3 ON (T1.subject=T3.subject) JOIN triples T4
ON (T1.subject=T4.subject) WHERE
T1.predicate = ‘ <http://www.w3.org/2000/01/rdf-schema#label’ and T2.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T3.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric2’
and T4.predicate = ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’;
```

SPARQL Query High-1Star: 1 star pattern, high multiplicity MVP (Feature ∼19)

```
select * where {
   ?prod bsbm:productFeature ?pf; rdfs:label ?prodLabel;
         bsbm:productPropertyNumeric1 ?num1; bsbm:productPropertyNumeric2 ?num2 . }
```

>> **Hive script.**

```
CREATE TABLE Result_high1(Sub1 String, Prop1 String, Obj1 String, Prop2 String,
Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String);
INSERT OVERWRITE TABLE Result_high1
SELECT T1.subject sub1, T1.predicate prop1, T1.object obj1, T2.predicate prop2,
T2.object obj2, T3.predicate prop3, T3.object obj3, T4.predicate prop4, T4.object
obj4 FROM triples T1 JOIN triples T2 ON (T1.subject = T2.subject) JOIN triples T3
ON (T1.subject=T3.subject) JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate = ‘ <http://www.w3.org/2000/01/rdf-schema#label’ and T2.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T3.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric2’
and T4.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’;
```

SPARQL Query Base2Star: 2 star patterns, no MV property (multiplicity 0)

```
select * where {
    ?offer bsbm:deliveryDays ?days; bsbm:vendor ?vendor; bsbm:price ?price;
          bsbm:product ?prod; bsbm:validTo ?validTo .
    ?prod bsbm:productPropertyNumeric3 ?num3; rdfs:label ?prodLabel;
          bsbm:productPropertyNumeric1 ?num1; bsbm:productPropertyNumeric2 ?num2 . }
```

SPARQL Query Low-2Star: 2 star patterns, low multiplicity MVP (rdf:type ∼6)

```
select * where {
    ?offer bsbm:deliveryDays ?days; bsbm:vendor ?vendor; bsbm:price ?price;
          bsbm:product ?prod; bsbm:validTo ?validTo .
    ?prod rdf:type ?pType; rdfs:label ?prodLabel;
          bsbm:productPropertyNumeric1 ?num1; bsbm:productPropertyNumeric2 ?num2 . }
```

>> Hive script.

```
CREATE TABLE Result_low2(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String,
Prop5 String, Obj5 String, Prop6 String, Obj6 String, Prop7 String, Obj7 String,
Prop8 String, Obj8 String, Prop9 String, Obj9 String);
INSERT OVERWRITE TABLE Result_low2
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2, J1.p3, J1.o3, J1.p4, J1.o4, J1.p5,
J1.o5, J2.p1, J2.o1, J2.p2, J2.o2, J2.p3, J2.o3, J2.p4, J2.o4 from
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3, T4.predicate p4, T4.object o4, T5.predicate p5,
T5.object o5 FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) JOIN
triples T3 ON (T1.subject=T3.subject) JOIN triples T4 ON (T1.subject=T4.subject)
JOIN triples T5 ON (T1.subject=T5.subject) WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/deliveryDays’
and T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’
and T3.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’
and T4.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/validTo’
and T5.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’) J1
JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3, T4.predicate p4, T4.object o4
FROM triples T1 JOIN triples T2 ON (T1.subject = T2.subject)
JOIN triples T3 ON (T1.subject=T3.subject) JOIN triples T4 ON (T1.subject=T4.subject)
WHERE T1.predicate = ‘ <http://www.w3.org/2000/01/rdf-schema#label’ and
T2.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T3.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric2’
and T4.predicate = ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’) J2
ON (trim(J1.o5)=trim(J2.s1));
```

SPARQL Query High-2Star: 2 star patterns, high multiplicity MVP (Feature ∼19)

```
select * where {
    ?offer bsbm:deliveryDays ?days; bsbm:vendor ?vendor; bsbm:price ?price;
          bsbm:product ?prod; ?offer bsbm:validTo ?validTo .
    ?prod bsbm:productFeature ?pf; rdfs:label ?prodLabel;
          bsbm:productPropertyNumeric1 ?num1; bsbm:productPropertyNumeric2 ?num2 . }
```

>> **Hive script.**

```
CREATE TABLE Result_hi2(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String,
Prop5 String, Obj5 String, Prop6 String, Obj6 String, Prop7 String, Obj7 String,
Prop8 String, Obj8 String, Prop9 String, Obj9 String);
INSERT OVERWRITE TABLE Result_hi2
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2, J1.p3, J1.o3, J1.p4, J1.o4, J1.p5,
J1.o5, J2.p1, J2.o1, J2.p2, J2.o2, J2.p3, J2.o3, J2.p4, J2.o4 from
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3, T4.predicate p4, T4.object o4,
T5.predicate p5, T5.object o5 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject)
JOIN triples T4 ON (T1.subject=T4.subject) JOIN triples T5 ON (T1.subject=T5.subject)
WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/deliveryDays’
and T2.predicate =‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’
and T3.predicate =‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’
and T4.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/validTo’
and T5.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3, T4.predicate p4, T4.object o4
FROM triples T1 JOIN triples T2 ON (T1.subject = T2.subject) JOIN triples T3
ON (T1.subject=T3.subject) JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate = ‘ <http://www.w3.org/2000/01/rdf-schema#label’ and T2.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T3.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric2’
```

```
and T4.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’) J2
ON (trim(J1.o5)=trim(J2.s1));
```

SPARQL Query mv-2p: Star-MVP with 2 triple patterns

```
select * where {
    ?a1 rdf:type bsbm:ProductFeature; rdfs:label ?pFLabel; dc:publisher ?apub .
    ?s1 bsbm:productFeature ?a1; dc:publisher ?producer .}
```

SPARQL Query mv-3p: Star-MVP with 3 triple patterns

```
select * where {
    ?a1 rdf:type bsbm:ProductFeature; rdfs:label ?pFLabel; dc:publisher ?apub .
    ?s1 bsbm:productFeature ?a1; dc:publisher ?producer;
        productPropertyNumeric1 ?pN1 .}
```

SPARQL Query mv-4p: Star-MVP with 4 triple patterns

```
select * where {
    ?a1 rdf:type bsbm:ProductFeature; rdfs:label ?pFLabel; dc:publisher ?apub .
    ?s1 bsbm:productFeature ?a1; dc:publisher ?producer;
        bsbm:productPropertyNumeric1 ?pN1; bsbm:productPropertyNumeric2 ?pN2 .}
```

SPARQL Query mv-5p: Star-MVP with 5 triple patterns

```
select * where {
    ?a1 rdf:type bsbm:ProductFeature; rdfs:label ?pFLabel; dc:publisher ?apub .
    ?s1 bsbm:productFeature ?a1; dc:publisher ?producer;
        bsbm:productPropertyNumeric1 ?pN1; bsbm:productPropertyNumeric2 ?pN2;
        bsbm:productPropertyNumeric3 ?pN3 .}
```

>> **Hive script.**

```
CREATE TABLE Result_mvp5(Sub1 String, Sub2 String, Prop1 String,
Obj1 String, Prop2 String, Obj2 String, Prop3 String, Obj3 String,
Prop4 String, Obj4 String, Prop5 String, Obj5 String, Prop6 String,
Obj6 String, Prop7 String, Obj7 String);
INSERT OVERWRITE TABLE Result_mvp5 select J1.s1, J2.s1, J1.p1, J1.o1,
J1.p2, J1.o2, J2.p1, J2.o1, J2.p2, J2.o2, J2.p3, J2.o3, J2.p4, J2.o4,
J2.p5, J2.o5 from
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
```

```
T2.object o2 FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject)
WHERE T1.predicate=‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’
and T1.object=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/ProductFeature
and T2.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3, T4.predicate p4, T4.object o4,
T5.predicate p5, T5.object o5 FROM triples T1 JOIN triples T2
ON (T1.subject = T2.subject)JOIN triples T3 ON (T1.subject=T3.subject
JOIN triples T4 ON (T1.subject=T4.subject) JOIN triples T5
ON (T1.subject=T5.subject) WHERE T1.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T2.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric2’
and T3.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric3’
and T4.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’
and T5.predicate=’ <http://purl.org/dc/elements/1.1/publisher) J2
ON (trim(J1.s1)=trim(J2.o4));
```

B.2 Queries on DBInfobox Dataset

Queries used have multiple multi-valued properties with varying multiplicity across instances. While all five queries involve an *MVJoin*, Dbp5 involves an object-object *MVJoin* where both joining triple patterns contain a multi-valued property ($influencedBy$).

SPARQL Query Dbp1: Retrieve writers of works and the people that they influenced.

```
select * where {
    ?work foaf:name ?wname; prop:writer ?writer; rdf:type prop:Work .
    ?writer foaf:name ?pname; prop:influenced ?another . }
```

SPARQL Query Dbp2: Retrieve details about movie and the starring actors.

```
select * where {
    ?film foaf:name ?fname; prop:starring ?actor; prop:director ?director .
          prop:producer ?producer .
```

```
    ?actor foaf:name ?aname; prop:birthPlace ?place . }
```

SPARQL Query Dbp3: Retrieve details about books and their authors.

```
select * where {
    ?book rdf:type prop:Book; foaf:name ?bname;
          prop:isbn ?bisbn; prop:author ?author .
    ?author rdf:type prop:Writer; foaf:name ?aname; prop:birthPlace ?place;
          prop:influencedBy ?another . }
```

SPARQL Query Dbp4: Retrieve Philosophers and other people they influenced.

```
select * where {
    ?phil rdf:type prop:Philosopher; prop:name ?name; prop:mainInterest ?interest;
          prop:influencedBy ?another .
    ?another rdf:type prop:Person; prop:name ?aname;
          prop:birthPlace ?birth; prop:influencedBy ?other . }
```

SPARQL Query Dbp5: Retrieve Philosophers and others influenced by their mentors.

```
select * where {
    ?phil rdf:type prop:Philosopher; prop:name ?name; prop:birthPlace ?place;
          prop:mainInterest ?interest; prop:influencedBy ?influencer .
    ?other rdf:type prop:Person; prop:name ?aname; prop:birthPlace ?birth;
          prop:influencedBy ?influencer . }
```

Appendix C

Evaluated RDF Unbound-Property Queries

C.1 Queries on Berlin SPARQL Benchmark Dataset

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

SPARQL Query B0 (Base: Two star subpatterns with no unbound properties).

```
SELECT * WHERE {
   ?offer bsbm:product ?prod; bsbm:vendor ?vend .
   ?prod bsbm:productFeature ?pf; bsbm:productPropertyNumeric1 ?pN1 . }
```

>> Hive script.

```
CREATE TABLE ResultQ9(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String);
INSERT OVERWRITE TABLE ResultQ9
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p3, J1.o3, J2.p1, J2.o1, J2.p4, J2.o4 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T3.predicate p3, T3.object o3
FROM triples T1 JOIN triples T3 ON (T1.subject=T3.subject) WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’) J1
JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T4.predicate p4,
```

```
T4.object o4 FROM triples T1 JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’
and T4.predicate =
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’)
J2 ON (TRIM(J1.s1)=TRIM(J2.o4));
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(’>’);
SPLIT A into propNum1 IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’,
product IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’,
prodFeat IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’,
vendor IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’;
star1 = Join vendor by $0, product by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6;
star2 = Join propNum1 by $0, prodFeat by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6;
joinResult = Join flatStar1 by TRIM($4), flatStar2 by TRIM($0);
store joinResult into ‘sparql_output’;
```

SPARQL Query B1 (Two star patterns with 1 unbound-property).

```
SELECT * WHERE {
   ?offer ?p ?prod; bsbm:vendor ?vend .
   ?prod bsbm:productFeature ?pf; bsbm:productPropertyNumeric1 ?pN1 . }
```

>> Hive script.

```
CREATE TABLE ResultQ9(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String);
INSERT OVERWRITE TABLE ResultQ9
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p3, J1.o3, J2.p1, J2.o1, J2.p4, J2.p4 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T3.predicate p3, T3.object
o3 FROM triples T1 JOIN triples T3 ON (T1.subject=T3.subject) WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’) J1
```

```
JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T4.predicate p4,
T4.object o4 FROM triples T1 JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’)
J2 ON (TRIM(J1.s1)=TRIM(J2.o4));
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
SPLIT A into propNum1 IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’,
prodFeat IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’,
vendor IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’;
star1 = Join vendor by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6;
star2 = Join propNum1 by $0, prodFeat by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6;
joinResult = Join flatStar1 by TRIM($4), flatStar2 by TRIM($0);
store joinResult into ‘sparql_output’;
```

SPARQL Query B2 (Query B1 with partially known object).

```
SELECT * WHERE {
   ?offer ?p ?obj; bsbm:vendor ?vend .
   FILTER regex(?obj, ‘‘Product’’)
   ?obj bsbm:productFeature ?pf; bsbm:productPropertyNumeric1 ?pN1 . }
```

>> Hive script.

```
CREATE TABLE ResultQ9(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String);
INSERT OVERWRITE TABLE ResultQ9
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p3, J1.o3, J2.p1, J2.o1, J2.p4, J2.p4 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T3.predicate p3, T3.object o3
FROM triples T1 JOIN triples T3 ON (T1.subject=T3.subject) WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’) J1
JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T4.predicate p4,
```

```
T4.object o4 FROM triples T1 JOIN triples T4 ON (T1.subject=T4.subject
and T4.object LIKE '%Product%') WHERE
T1.predicate=' <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor')
J2 ON (TRIM(J1.s1)=TRIM(J2.o4));
```

>> Pig script.

```
A = LOAD 'sparql_source' using PigStorage('>');
SPLIT A into propNum1 IF $1 eq
' <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1',
prodFeat IF $1 eq
' <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature',
vendor IF $1 eq ' <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor',
relationToProduct IF ($2 matches '.*Product.*');
star1 = Join vendor by $0, relationToProduct by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6;
star2 = Join propNum1 by $0, prodFeat by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6;
joinResult = Join flatStar1 by TRIM($4), flatStar2 by TRIM($0);
store joinResult into 'sparql_output';
```

SPARQL Query B3 (Two star subpatterns with 2 unbound-property triple patterns, one with partially known object).

```
SELECT * WHERE {
   ?offer ?p ?obj; bsbm:vendor ?vend; ?p1 ?smthng .
   FILTER regex(?smthng, "date")
   ?obj bsbm:productFeature ?pf; bsbm:productPropertyNumeric1 ?pN1 . }
```

SPARQL Query B4 (Two star subpatterns, 1 unbound-property to retrieve all we know about Offer's products).

```
SELECT * WHERE {
   ?offer bsbm:product ?prod; bsbm:vendor ?vend .
   ?prod bsbm:productPropertyNumeric1 ?pN1; ?p ?anything . }
```

>> Hive script.

```
CREATE TABLE ResultQ9(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String);
INSERT OVERWRITE TABLE ResultQ9
```

```
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p3, J1.o3, J2.p1, J2.o1, J2.p4, J2.o4 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T3.predicate p3, T3.object o3
FROM triples T1 JOIN triples T3 ON (T1.subject=T3.subject) WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’)
J1 JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T4.predicate p4,
T4.object o4 FROM triples T1 JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’ and
T4.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’)
J2 ON (TRIM(J1.s1)=TRIM(J2.o4));
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
SPLIT A into propNum1 IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’,
product IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product’,
vendor IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’;
star1 = Join vendor by $0, product by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6;
star2 = Join propNum1 by $0, A1 by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6;
joinResult = Join flatStar1 by TRIM($4), flatStar2 by TRIM($0);
store joinResult into ‘sparql_output’;
```

SPARQL Query B1-4bnd (Varying size of bound component - 4 bound patterns).

```
SELECT * WHERE {
   ?prod bsbm:productFeature ?pf; bsbm:productPropertyNumeric1 ?pN1;
         bsbm:productPropertyNumeric3 ?pN3 .
   ?offer ?p ?prod; bsbm:vendor ?vend . }
```

>> Hive script.

```
CREATE TABLE ResultQ9(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String,
Prop5 String, Obj5 String);
INSERT OVERWRITE TABLE ResultQ9 select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2,
J1.p3, J1.o3, J2.p1, J2.o1, J2.p4, J2.p4 FROM
```

```
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject)
WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T2.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric3’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’)
J1 JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T4.predicate p4,
T4.object o4 FROM triples T1 JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’)
J2 ON (TRIM(J1.s1)=TRIM(J2.o4));
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
SPLIT A into propNum1 IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’,
propNum3 IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric3’,
prodFeat IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’,
vendor IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’;
star1 = Join vendor by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6;
star2 = Join propNum1 by $0, propNum3 by $0, prodFeat by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6, $9, $10;
joinResult = Join flatStar1 by TRIM($4), flatStar2 by TRIM($0);
store joinResult into ‘sparql_output’;
```

SPARQL Query B1-5bnd (Varying size of bound component - 5 bound patterns).

```
SELECT * WHERE {
   ?prod bsbm:productFeature ?pf; bsbm:productPropertyNumeric1 ?pN1;
         bsbm:productPropertyNumeric3 ?pN3 .
   ?offer ?p ?prod; bsbm:vendor ?vend; bsbm:price ?price . }
```

>> **Hive script.**

```
CREATE TABLE ResultQ9(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String,
Prop5 String, Obj5 String, Prop6 String, Obj6 String);
INSERT OVERWRITE TABLE ResultQ9 select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2,
J1.p3, J1.o3, J2.p1, J2.o1, J2.p2, J2.o2, J2.p4, J2.p4 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2 ON
(T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T2.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric3’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T4.predicate p4, T4.object o4 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’ and
T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’) J2
ON (TRIM(J1.s1)=TRIM(J2.o4));
```

>> **Pig script.**

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
SPLIT A into propNum1 IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’,
propNum3 IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric3’,
prodFeat IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’,
price IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’,
vendor IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’;
star1 = Join vendor by $0, price by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6, $9, $10;
star2 = Join propNum1 by $0, propNum3 by $0, prodFeat by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6, $9, $10;
joinResult = Join flatStar1 by TRIM($6), flatStar2 by TRIM($0);
```

```
store joinResult into ‘sparql_output’;
```

SPARQL Query B1-6bnd (Varying size of bound component - 6 bound patterns).

```
SELECT * WHERE {
   ?prod bsbm:productFeature ?pf; bsbm:productPropertyNumeric1 ?pN1;
         bsbm:productPropertyNumeric3 ?pN3 .
   ?offer ?p ?prod; bsbm:vendor ?vend; bsbm:price ?price; bsbm:validFrom ?vFrom . }
```

>> Hive script.

```
CREATE TABLE ResultQ9(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String,
Prop5 String, Obj5 String, Prop6 String, Obj6 String, Prop7 String, Obj7 String);
INSERT OVERWRITE TABLE ResultQ9 select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2,
J1.p3, J1.o3, J2.p1, J2.o1, J2.p2, J2.o2, J2.p3, J2.o3, J2.p4, J2.p4 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2 ON
(T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject)
WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
and T2.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric3’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3, T4.predicate p4, T4.object o4 FROM triples T1
JOIN triples T2 ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject)
JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’ and
T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’ and
T3.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/validFrom’)
J2 ON (TRIM(J1.s1)=TRIM(J2.o4));
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
SPLIT A into propNum1 IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’,
```

```
propNum3 IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric3’,
prodFeat IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’,
price IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’,
vendor IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’,
validFrom IF $1 eq
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/validFrom’;
star1 = Join vendor by $0, price by $0, validFrom by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6, $9, $10, $13, $14;
star2 = Join propNum1 by $0, propNum3 by $0, prodFeat by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6, $9, $10;
joinResult = Join flatStar1 by TRIM($8), flatStar2 by TRIM($0);
store joinResult into ‘sparql_output’;
```

SPARQL Query B5 (Two star subpatterns with 2 unbound-properties).

```
SELECT * WHERE {
   ?offer ?p ?prod; bsbm:vendor ?vend; bsbm:price ?price .
   ?prod rdf:type bsbm:Product; ?p1 ?allobjs . }
```

>> Hive script.

```
CREATE TABLE ResultQ2_2unb(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String,
Prop5 String, Obj5 String);
INSERT OVERWRITE TABLE ResultQ2_2unb select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2,
J1.o2, J2.p1, J2.o1, J2.p2, J2.o2, J2.p4, J2.p4 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2
FROM triples T1 JOIN triples T2 ON (T1.subject = T2.subject)
WHERE T1.predicate = ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’
and T1.object=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/Product’
) J1 JOIN (SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T4.predicate p4, T4.object o4 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T4 ON (T1.subject=T4.subject)
WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’ and
T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’) J2
ON (TRIM(J1.s1)=TRIM(J2.o4));
```

>> **Pig script.**

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
A2 = LOAD ‘sparql_source’ using PigStorage(‘>’);
SPLIT A into typeProd IF $1 eq ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’
and $2 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/Product’,
price IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’,
vendor IF $1 eq ‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’;
star1 = Join vendor by $0, price by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6, $9, $10;
star2 = Join typeProd by $0, A2 by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6;
joinResult = Join flatStar1 by TRIM($6), flatStar2 by TRIM($0);
store joinResult into ‘sparql_output’;
```

SPARQL Query B6 (Three star subpatterns with 1 unbound property.

```
SELECT * WHERE {
   ?prod bsbm:productFeature ?pf; bsbm:productPropertyNumeric1 ?pN1;
         bsbm:productPropertyNumeric3 ?pN3 .
   ?offer ?p ?prod; bsbm:vendor ?vend; bsbm:price ?price; bsbm:validFrom ?vFrom .
   ?vend foaf:homepage ?vHpage; bsbm:country ?vCountry; rdfs:label ?vLabel . }
```

>> **Hive script.**

```
CREATE TABLE ResultQ3(Sub1 String, Sub2 String, Sub3 String, Prop1 String,
Obj1 String, Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String,
Obj4 String, Prop5 String, Obj5 String, Prop6 String, Obj6 String, Prop7 String,
Obj7 String, Prop8 String, Obj8 String, Prop9 String, Obj9 String, Prop10 String,
Obj10 String);
INSERT OVERWRITE TABLE ResultQ3
select J1.s1, J2.s1, J3.s1, J1.p1, J1.o1, J1.p2, J1.o2, J1.p3, J1.o3, J2.p1, J2.o1,
J2.p2, J2.o2, J2.p3, J2.o3, J2.p4, J2.p4, J3.p1, J3.o1, J3.p2, J3.o2, J3.p3, J3.o3
FROM ( SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject)
WHERE T1.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1’
```

```
and T2.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric3’
and T3.predicate=
‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3, T4.predicate p4, T4.object o4 FROM triples T1
JOIN triples T2 ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject
JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor’ and
T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price’ and
T3.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/validFrom’)
J2 ON (TRIM(J1.s1)=TRIM(J2.o4)) JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2 ON
(T1.subject = T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=’ <http://xmlns.com/foaf/0.1/homepage’ and
T2.predicate=‘ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/country’
and T3.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’) J3
ON (TRIM(J2.o1)=TRIM(J3.s1));
```

C.2 Real-world Queries on Bio2RDF Dataset

```
PREFIX bio: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

SPARQL Query A1 (What is known about hexokinase).

```
SELECT * WHERE {
   ?s1 rdf:type ?type1; rdfs:label ?label1; ?p1 ?o1 .
   FILTER regex(?o1, ‘‘hexokinase’’) }
```

>> Hive script.

```
CREATE TABLE Hex1(Sub1 String, Prop1 String, Obj1 String, Prop2 String,
Obj2 String, Prop3 String, Obj3 String);
INSERT OVERWRITE TABLE Hex1
SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
```

```
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject and T1.object LIKE '%hexokinase%')
JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T2.predicate=' <http://www.w3.org/1999/02/22-rdf-syntax-ns#type' and
T3.predicate=' <http://www.w3.org/2000/01/rdf-schema#label';
```

>> Pig script.

```
A = LOAD 'sparql_source' using PigStorage('>');
SPLIT A into label IF $1 eq ' <http://www.w3.org/2000/01/rdf-schema#label',
type IF $1 eq ' <http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
hex IF ($2 matches '.*hexokinase.*');
star1 = Join hex by $0, type by $0, label by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $4, $5, $8, $9;
store flatStar1 into 'sparql_output';
```

SPARQL Query A2 (What is known about hexokinase gene, HK1).

```
SELECT * WHERE {
   ?s1 rdf:type ?type1; rdfs:label ?label1; ?p1 ?o1 .
   FILTER regex(?o1, ''HK1'') }
```

>> Hive script.

```
CREATE TABLE Hex1(Sub1 String, Prop1 String, Obj1 String, Prop2 String,
Obj2 String, Prop3 String, Obj3 String);
INSERT OVERWRITE TABLE Hex1
SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject and T1.object LIKE '%HK1%')
JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T2.predicate=' <http://www.w3.org/1999/02/22-rdf-syntax-ns#type' and
T3.predicate=' <http://www.w3.org/2000/01/rdf-schema#label';
```

>> Pig script.

```
A = LOAD 'sparql_source' using PigStorage('>');
SPLIT A into label IF $1 eq ' <http://www.w3.org/2000/01/rdf-schema#label',
type IF $1 eq ' <http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
hk1 IF ($2 matches '.*HK1.*');
star1 = Join hk1 by $0, type by $0, label by $0;
```

```
flatStar1 = FOREACH star1 generate $0, $1, $2, $4, $5, $8, $9;
store flatStar1 into ‘sparql_output’;
```

SPARQL Query A3 (Analyse Parkinson - GO term distribution).

```
SELECT * WHERE {
   ?s1 rdfs:label ?label1; ?p1 ?o1 .
   FILTER regex(?o1, ’’rxr’’)
   ?s2 ?p2 ?s1; bio:xGO ?go . }
```

>> Hive script.

```
CREATE TABLE Park(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String);
INSERT OVERWRITE TABLE Park
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2, J2.p1, J2.o1, J2.p2, J2.o2 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) WHERE
T2.predicate=‘ <http://bio2rdf.org/ns/bio2rdf#xGO’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) WHERE T1.object
LIKE ‘%rxr%’ and T2.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’)J2
ON (TRIM(J1.o1)=TRIM(J2.s1));
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
SPLIT A into label IF $1 eq ‘ <http://www.w3.org/2000/01/rdf-schema#label’,
rxr IF ($2 matches ‘.*rxr.*’),
xGO IF $1 eq ‘ <http://bio2rdf.org/ns/bio2rdf#xGO’;
star1 = Join rxr by $0, label by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $4, $5;
star2 = Join xGO by $0, A1 by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6;
joinResult = Join flatStar2 by TRIM($4), flatStar1 by TRIM($0);
store joinResult into ‘sparql_output’;
```

SPARQL Query A4 (Analyse Parkinson - GO term distribution).

```
SELECT * WHERE {
```

```
   ?s1 rdfs:label ?label1; ?p1 ?o1 .
   FILTER regex(?o1, ''rxr'')
   ?s2 bio:xGO ?go; rdfs:label ?label2; ?p2 ?s1 . }
```

>> Hive script.

```
CREATE TABLE Park(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String,
Prop5 String, Obj5 String);
INSERT OVERWRITE TABLE Park select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2,
J1.p3, J1.o3, J2.p1, J2.o1, J2.p2, J2.o2 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T2.predicate=‘ <http://bio2rdf.org/ns/bio2rdf#xGO’
and T3.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) WHERE T1.object
LIKE ‘%rxr%’ and T2.predicate=‘ <http://www.w3.org/2000/01/rdf-schema#label’)J2
ON (TRIM(J1.o1)=TRIM(J2.s1));
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
SPLIT A into label IF $1 eq ‘ <http://www.w3.org/2000/01/rdf-schema#label’,
rxr IF ($2 matches ‘.*rxr.*’),
xGO IF $1 eq ‘ <http://bio2rdf.org/ns/bio2rdf#xGO’;
star1 = Join rxr by $0, label by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $4, $5;
star2 = Join xGO by $0, label by $0, A1 by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6, $8, $9;
joinResult = Join flatStar2 by TRIM($6), flatStar1 by TRIM($0);
store joinResult into ‘sparql_output’;
```

SPARQL Query A5 (Gene GO annotation for nur77).

```
SELECT * WHERE {
   ?s1 rdfs:label ?label1; ?p1 ?o1; p2 ?o2 .
   FILTER regex(?o1, ''nur77'')
```

```
   ?o2 rdfs:label ?label2 .}
```

>> Hive script.

```
CREATE TABLE nur77(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String);
INSERT OVERWRITE TABLE nur77
SELECT J1.s1, T4.subject, J1.p1, J1.o1, J1.p2, J1.o2, J1.p3, J1.o3,
T4.predicate, T4.object FROM (
SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2
ON (T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.object LIKE '%nur77%' and
T2.predicate=' <http://www.w3.org/2000/01/rdf-schema#label')
J1 JOIN triples T4 ON (TRIM(J1.o3)=TRIM(T4.subject))
where T4.predicate=' <http://www.w3.org/2000/01/rdf-schema#label';
```

>> Pig script.

```
A = LOAD 'sparql_source' using PigStorage('>');
A1 = LOAD 'sparql_source' using PigStorage('>');
SPLIT A into label IF $1 eq ' <http://www.w3.org/2000/01/rdf-schema#label',
nur77 IF ($2 matches '.*nur77.*');
star1 = Join nur77 by $0, label by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $4, $5, $7, $8;
joinResult = Join flatStar1 by TRIM($6), label by TRIM($0);
store joinResult into 'sparql_output';
```

SPARQL Query A6 (Look for Hexokinase in GO or MeSH controlled vocabulary).

```
SELECT * WHERE {
   ?s1 rdf:type ?type1; rdfs:label ?label1 .
   ?s1 rdfs:comment ?comm1 ; bio:isA ?s2; ?p1 ?o1 .
   FILTER regex(?o1, ''hexokinase'')
   ?s2 rdfs:label ?label2; rdfs:comment ?comm2 . }
```

>> Hive script.

```
CREATE TABLE Park(Sub1 String, Sub2 String, Prop1 String, Obj1 String, Prop2 String,
Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String, Prop5 String,
Obj5 String, Prop6 String, Obj6 String, Prop7 String, Obj7 String);
```

```
INSERT OVERWRITE TABLE Park select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2,
J1.p3, J1.o3, J1.p4, J1.o4, J1.p5, J1.o5, J2.p1, J2.o1, J2.p2, J2.o2 FROM
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3, T4.predicate p4, T4.object o4, T5.predicate p5,
T2.object o5 FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) JOIN
triples T3 ON (T1.subject=T3.subject) JOIN triples T4 ON (T1.subject=T4.subject) JOIN
triples T5 ON (T1.subject=T5.subject) WHERE T1.object LIKE '%hexokinase%'
and T2.predicate=' <http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
and T3.predicate=' <http://www.w3.org/2000/01/rdf-schema#comment'
and T4.predicate=' <http://www.w3.org/2000/01/rdf-schema#label'
and T5.predicate=' <http://bio2rdf.org/ns/bio2rdf#isA')J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) WHERE
T1.predicate=' <http://www.w3.org/2000/01/rdf-schema#comment' and T2.predicate=
' <http://www.w3.org/2000/01/rdf-schema#label')J2 ON (TRIM(J1.o5)=TRIM(J2.s1));
```

>> Pig script.

```
A = LOAD 'sparql_source' using PigStorage('>');
SPLIT A into label IF $1 eq ' <http://www.w3.org/2000/01/rdf-schema#label',
comment IF $1 eq ' <http://www.w3.org/2000/01/rdf-schema#comment',
type IF $1 eq ' <http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
isA IF $1 eq ' <http://bio2rdf.org/ns/bio2rdf#isA',
hexo IF ($2 matches '.*hexokinase.*');
star1 = Join hexo by $0, type by $0, comment by $0, label by $0, isA by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $4, $5, $8, $9, $11, $12, $14, $15;
star2 = Join comment by $0, label by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $4, $5;
joinResult = Join flatStar1 by TRIM($10), flatStar2 by TRIM($0);
store joinResult into 'sparql_output';
```

C.3 Real-world Queries on DBInfoBox and Billion Triple Challenge Datasets

```
PREFIX db: <http://dbpedia.org/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

SPARQL Query C1 (What information do we have about Scientists?).

```
SELECT * WHERE {
    ?s1 rdf:type db:ontology/Scientist; ?p ?o . }
```

>> Hive script.

```
CREATE TABLE Result_Q0a(Sub1 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String);
INSERT OVERWRITE TABLE Result_Q0a
SELECT T1.subject sub1, T1.predicate prop1, T1.object obj1, T2.predicate prop2,
T2.object obj2 FROM triples T1 JOIN triples T2 ON (T1.subject = T2.subject)
WHERE T1.predicate = ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’
and T1.object=‘ <http://dbpedia.org/ontology/Scientist’;
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
offerType = FILTER A by $1 eq ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’
and $2 eq ‘ <http://dbpedia.org/ontology/Scientist’;
star1 = join offerType by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6;
store flatStar1 into ‘sparql_output’;
```

SPARQL Query C2 (What do we know about the Sopranos series?).

```
SELECT * WHERE {
    ?s1 db:ontology/series db:resource/The_Sopranos; ?p ?o . }
```

>> Hive script.

```
CREATE TABLE Result_Q0a(Sub1 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String);
INSERT OVERWRITE TABLE Result_Q0a
SELECT T1.subject sub1, T1.predicate prop1, T1.object obj1, T2.predicate prop2,
T2.object obj2 FROM triples T1 JOIN triples T2 ON (T1.subject = T2.subject)
WHERE T1.predicate = ‘ <http://dbpedia.org/ontology/series’
and T1.object=‘ <http://dbpedia.org/resource/The_Sopranos’;
```

>> **Pig script.**

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘>’);
offerType = FILTER A by $1 eq ‘ <http://dbpedia.org/ontology/series’ and
$2 eq ‘ <http://dbpedia.org/resource/The_Sopranos’;
star1 = join offerType by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6;
store flatStar1 into ‘sparql_output’;
```

SPARQL Query C3 (Persons in some way related to books).

```
SELECT * WHERE {
    ?book rdf:type db:Book; foaf:name ?bname; db:isbn ?isbn; ?p ?person .
    ?person rdf:type db:Writer; foaf:name ?aname; db:birthPlace ?place . }
```

>> **Hive script.**

```
CREATE TABLE Res_dbp2(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String,
Prop5 String, Obj5 String, Prop6 String, Obj6 String, Prop7 String, Obj7 String);
INSERT OVERWRITE TABLE Res_dbp2 select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2,
J1.p3, J1.o3, J1.p4, J1.o4, J2.p1, J2.o1, J2.p2, J2.o2, J2.p3, J2.o3 from
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2,
T2.object o2, T3.predicate p3, T3.object o3, T4.predicate p4, T4.object o4 FROM
triples T1 JOIN triples T2 ON (T1.subject=T2.subject) JOIN triples T3
ON (T1.subject=T3.subject) JOIN triples T4 ON (T1.subject=T4.subject) WHERE
T1.predicate=‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’ and
T1.object=‘ <http://dbpedia.org/ontology/Book’ and
T2.predicate=‘ <http://xmlns.com/foaf/0.1/name’ and
T3.predicate=‘ <http://dbpedia.org/ontology/isbn’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2,
T3.predicate p3, T3.object o3 FROM triples T1 JOIN triples T2 ON
(T1.subject=T2.subject) JOIN triples T3 ON (T1.subject=T3.subject) WHERE
T1.predicate=‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’
and T1.object=‘ <http://dbpedia.org/ontology/Writer’ and
T2.predicate=‘ <http://xmlns.com/foaf/0.1/name’ and
T3.predicate=‘ <http://dbpedia.org/ontology/birthPlace’) J2
ON (trim(J1.o4)=trim(J2.s1));
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(‘<’);
A1 = LOAD ‘sparql_source’ using PigStorage(‘<’);
SPLIT A into name IF $1 eq ‘ <http://xmlns.com/foaf/0.1/name’,
typeBook IF $1 eq ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’
and $2 eq ‘ <http://dbpedia.org/ontology/Book’,
typeWriter IF $1 eq ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’
and $2 eq ‘ <http://dbpedia.org/ontology/Writer’,
birthPlace IF $1 eq ‘ <http://dbpedia.org/ontology/birthPlace’,
isbn IF $1 eq ‘ <http://dbpedia.org/ontology/isbn’;
star1 = Join typeBook by $0, name by $0, isbn by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6, $8, $9, $11, $12;
star2 = Join typeWriter by $0, name by $0, birthPlace by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6, $8, $9;
joinResult = Join flatStar2 by $0, flatStar1 by TRIM($8);
store joinResult into ’sparql_output’;
```

SPARQL Query C4 (What do we know about writers of books?).

```
SELECT * WHERE {
   ?book rdf:type prop:Book; foaf:name ?bname; prop:isbn ?isbn; ?p ?person .
   ?person rdf:type prop:Writer; ?p1 ?anyprop . }
```

>> Hive script.

```
CREATE TABLE Res_dbp2b(Sub1 String, Sub2 String, Prop1 String, Obj1 String,
Prop2 String, Obj2 String, Prop3 String, Obj3 String, Prop4 String, Obj4 String,
Prop5 String, Obj5 String, Prop6 String, Obj6 String);
INSERT OVERWRITE TABLE Res_dbp2b
select J1.s1, J2.s1, J1.p1, J1.o1, J1.p2, J1.o2, J1.p3, J1.o3, J1.p4, J1.o4, J2.p1,
J2.o1, J2.p2, J2.o2 from (SELECT T1.subject s1, T1.predicate p1, T1.object o1,
T2.predicate p2, T2.object o2, T3.predicate p3, T3.object o3, T4.predicate p4,
T4.object o4 FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) JOIN
triples T3 ON (T1.subject=T3.subject) JOIN triples T4 ON (T1.subject=T4.subject)
WHERE T1.predicate=‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’ and
T1.object=‘ <http://dbpedia.org/ontology/Book’
and T2.predicate=’ <http://xmlns.com/foaf/0.1/name’ and
T3.predicate=‘ <http://dbpedia.org/ontology/isbn’) J1 JOIN
(SELECT T1.subject s1, T1.predicate p1, T1.object o1, T2.predicate p2, T2.object o2
```

```
FROM triples T1 JOIN triples T2 ON (T1.subject=T2.subject) WHERE
T1.predicate=‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’ and
T1.object=‘ <http://dbpedia.org/ontology/Writer’) J2 ON (trim(J1.o4)=trim(J2.s1));
```

>> Pig script.

```
A = LOAD ‘sparql_source’ using PigStorage(‘>’);
A1 = LOAD ‘sparql_source’ using PigStorage(’>’);
A2 = LOAD ‘sparql_source’ using PigStorage(’>’);
SPLIT A into name IF $1 eq ’ <http://xmlns.com/foaf/0.1/name’,
typeBook IF $1 eq ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’ and
$2 eq ‘ <http://dbpedia.org/ontology/Book’,
typeWriter IF $1 eq ‘ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type’ and
$2 eq ‘ <http://dbpedia.org/ontology/Writer’,
isbn IF $1 eq ‘ <http://dbpedia.org/ontology/isbn’;
star1 = Join typeBook by $0, name by $0, isbn by $0, A1 by $0;
flatStar1 = FOREACH star1 generate $0, $1, $2, $5, $6, $8, $9, $11, $12;
star2 = Join typeWriter by $0, A2 by $0;
flatStar2 = FOREACH star2 generate $0, $1, $2, $5, $6;
joinResult = Join flatStar2 by $0, flatStar1 by TRIM($8);
store joinResult into ‘sparql_output’;
```

Appendix D

Evaluated RDF Analytical Queries

Hive Pre-processing. The Hive scripts in this section assume a pre-processing phase with vertically partitioned relations. Below, we provide a sample for the BSBM dataset.

```
SET hive.default.fileformat=Orc;
// Step 1: Create a Triples Relation (Subject, Predicate, Object)
Create external table triples (subject string, predicate string, object string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '>'
STORED AS TEXTFILE LOCATION '/user/hadoop/sparql_source';
//Step 2: Prefix-substitution
CREATE TABLE TempVP(Sub String, Prop String, Obj String);
INSERT OVERWRITE TABLE TempVP
select regexp_replace(regexp_replace(regexp_replace(regexp_replace
(regexp_replace(regexp_replace(regexp_replace(
subject, 'http://www.w3.org/1999/02/22-rdf-syntax-ns#', 'rdf:'),
'http://www.w3.org/2000/01/rdf-schema#', 'rdfs:'),
'http://xmlns.com/foaf/0.1/', 'fo:'),
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/', 'bsbm:'),
'http://purl.org/dc/elements/1.1/', 'dc:'),
'http://purl.org/stuff/rev#', 'rv:'),
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/', 'inst:'),
regexp_replace(regexp_replace(regexp_replace(regexp_replace
(regexp_replace(regexp_replace(regexp_replace(
predicate, 'http://www.w3.org/1999/02/22-rdf-syntax-ns#', 'rdf:'),
'http://www.w3.org/2000/01/rdf-schema#', 'rdfs:'),
'http://xmlns.com/foaf/0.1/', 'fo:'),
```

```
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/', 'bsbm:'),
'http://purl.org/dc/elements/1.1/', 'dc:'),
'http://purl.org/stuff/rev#', 'rv:'),
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/', 'inst:'),
regexp_replace(regexp_replace(regexp_replace(regexp_replace(
regexp_replace(regexp_replace(regexp_replace(
object, 'http://www.w3.org/1999/02/22-rdf-syntax-ns#', 'rdf:'),
'http://www.w3.org/2000/01/rdf-schema#', 'rdfs:'),
'http://xmlns.com/foaf/0.1/', 'fo:'),
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/', 'bsbm:'),
'http://purl.org/dc/elements/1.1/', 'dc:'),
'http://purl.org/stuff/rev#', 'rv:'),
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/', 'inst:'
) from triples;
```

//Step 3: Create vertically-partitioned relations

```
CREATE TABLE product(Sub string, Obj string) STORED as ORC;
INSERT OVERWRITE TABLE product
select Sub, Obj from TempVP where prop LIKE ''%bsbm:product";
CREATE TABLE typeAll(Sub string, Obj string) STORED as ORC;
INSERT OVERWRITE TABLE typeAll
select Sub, Obj from TempVP where prop LIKE ''%rdf:type";
CREATE TABLE typeProductType1(Sub string) STORED as ORC;
INSERT OVERWRITE TABLE typeProductType1
select Sub from typeAll where Obj LIKE '%<inst:ProductType1';
```

D.1 Queries on Berlin SPARQL Benchmark Dataset

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

SPARQL Query G1.

```
SELECT (COUNT(?price) As ?countTotal) (SUM(?price) As ?sumTotal) {
   ?prod rdf:type bsbm-inst:ProductType1; rdfs:label ?lab1 .
   ?offer bsbm:product ?prod; bsbm:price ?price . }
```

>> Hive (Naive) script.

```
CREATE TABLE q0_ALL (cntT String, sumT String);
INSERT OVERWRITE TABLE q0_ALL
select COUNT(G1.o4) cntT, SUM(regexp_extract(G1.o4,'"([0-9.]*)"(.*)',1)) sumT FROM
(SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(SELECT sub s1, 'bsbm:ProductType1' o1 FROM typeProductType1) T1 JOIN
(SELECT sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1)))G1;
```

SPARQL Query G2.

```
SELECT (COUNT(?price) As ?countTotal) (SUM(?price) As ?sumTotal) {
   ?prod rdf:type bsbm-inst:ProductType9; rdfs:label ?lab1 .
   ?offer bsbm:product ?prod; bsbm:price ?price . }
```

>> Hive (Naive) script.

```
CREATE TABLE q0_ALL (cntT String, sumT String);
INSERT OVERWRITE TABLE q0_ALL
select COUNT(G1.o4) cntT, SUM(regexp_extract(G1.o4,'"([0-9.]*)"(.*)',1)) sumT FROM
(SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(SELECT sub s1, 'bsbm:ProductType9' o1 FROM typeProductType1) T1 JOIN
(SELECT sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1)))G1;
```

SPARQL Query G3.

```
SELECT ?feature (COUNT(?price2) As ?countF) (SUM(?price2) As ?sumF){
   ?prod2 rdf:type bsbm-inst:ProductType1;
          rdfs:label ?lab2; bsbm:productFeature ?feature .
   ?offer2 bsbm:product ?prod2; bsbm:price ?price2 . }
GROUP BY ?feature
```

>> Hive (Naive) script.

```
CREATE TABLE q0_feat (feature String, cntF String, sumF String);
INSERT OVERWRITE TABLE q0_feat
select G2.o3, COUNT(G2.o5), SUM(regexp_extract(G2.o5,‘"([0-9.]*)"(.*)’,1)) FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4, SJ2.o2 o5
FROM( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM
(select sub s1, ’bsbm:ProductType1’ o1 FROM typeProductType1) T1 JOIN
(select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) JOIN
(select sub s1, obj o1 FROM productFeature)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1))) G2 GROUP BY G2.o3;
```

SPARQL Query G4.

```
SELECT ?feature (COUNT(?price2) As ?countF) (SUM(?price2) As ?sumF){
   ?prod2 rdf:type bsbm-inst:ProductType9;
          rdfs:label ?lab2; bsbm:productFeature ?feature .
   ?offer2 bsbm:product ?prod2; bsbm:price ?price2 . }
GROUP BY ?feature
```

>> Hive (Naive) script.

```
CREATE TABLE q0_feat (feature String, cntF String, sumF String);
INSERT OVERWRITE TABLE q0_feat
select G2.o3, COUNT(G2.o5), SUM(regexp_extract(G2.o5,‘"([0-9.]*)"(.*)’,1)) FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4, SJ2.o2 o5
FROM( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM
(select sub s1, ’bsbm:ProductType9’ o1 FROM typeProductType9) T1 JOIN
(select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) JOIN
(select sub s1, obj o1 FROM productFeature)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1))) G2 GROUP BY G2.o3;
```

SPARQL Query MG1.

```
SELECT ?feature ?sumF ?countF ?sumTotal ?countTotal {
{ SELECT ?feature (COUNT(?price2) As ?countF) (SUM(?price2) As ?sumF)
  { ?prod2 rdf:type bsbm-inst:ProductType1;
           rdfs:label ?lab2; bsbm:productFeature ?feature .
    ?offer2 bsbm:product ?prod2; bsbm:price ?price2 .
  } GROUP BY ?feature
}
{ SELECT (COUNT(?price) As ?countTotal) (SUM(?price) As ?sumTotal)
   { ?prod rdf:type bsbm-inst:ProductType1;
           rdfs:label ?lab1 .
    ?offer bsbm:product ?prod; bsbm:price ?price .
  }
} }
```

>> Hive (Naive) script.

```
CREATE TABLE q4 (feature String, cntT INT, sumT Double, cntF INT, sumF Double);
INSERT OVERWRITE TABLE q4
SELECT GP2.feat, GP1.cnT, GP1.smT, GP2.cnF, GP2.smF FROM
(select COUNT(G1.o4) cnT, SUM(regexp_extract(G1.o4,'"([0-9.]*)"(.*)',1)) smT FROM
(SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(SELECT sub s1, 'bsbm:ProductType1' o1 FROM typeProductType1) T1 JOIN
(SELECT sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1))) G1 ) GP1 JOIN (
select G2.o3 feat, COUNT(G2.o5) cnF,
SUM(regexp_extract(G2.o5,'"([0-9.]*)"(.*)',1)) smF
FROM( SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4,
SJ2.o2 o5 FROM( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM
(select sub s1, 'bsbm:ProductType1' o1 FROM typeProductType1) T1 JOIN (
select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) JOIN
(select sub s1, obj o1 FROM productFeature)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2 ON
```

```
(TRIM(SJ1.s1)=TRIM(SJ2.o1))) G2 GROUP BY G2.o3 ) GP2;
```

>> Hive MQO script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String, o5 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4, SJ2.o2 o5 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM
(select sub s1, 'bsbm:ProductType1' o1 FROM typeProductType1) T1 JOIN
(select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) LEFT OUTER JOIN
(select sub s1, obj o1 FROM productFeature)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1));

CREATE TABLE q4 (feat String, cntT INT, sumT Double, cntF INT, sumF Double);
INSERT OVERWRITE TABLE q4
select GP2.ft, GP2.cnF, GP2.smF, GP1.cnT, GP1.smT FROM
(select COUNT(G1.price) cnT, SUM(regexp_extract(G1.price,‘"([0-9.]*)"(.*)’,1)) smT
FROM (select distinct o1, o2, o4, o5 price from L1) G1 ) GP1 JOIN
(select G2.ft ft, COUNT(G2.pc) cnF,
SUM(regexp_extract(G2.pc,‘"([0-9.]*)"(.*)’,1)) smF
FROM( select o1, o2, o3 ft, o4, o5 pc from L1 where o5 is not NULL) G2
GROUP BY G2.feature) GP2;
```

SPARQL Query MG2.

```
SELECT ?feature ?sumF ?countF ?sumTotal ?countTotal {
{ SELECT ?feature (COUNT(?price2) As ?countF) (SUM(?price2) As ?sumF)
  { ?prod2 rdf:type bsbm-inst:ProductType9;
           rdfs:label ?lab2; bsbm:productFeature ?feature .
    ?offer2 bsbm:product ?prod2; bsbm:price ?price2 .
  } GROUP BY ?feature
}
{ SELECT (COUNT(?price) As ?countTotal) (SUM(?price) As ?sumTotal)
   { ?prod rdf:type bsbm-inst:ProductType9;
           rdfs:label ?lab1 .
    ?offer bsbm:product ?prod; bsbm:price ?price .
```

```
  }
} }
```

>> Hive (Naive) script.

```
CREATE TABLE q4 (feature String, cntT INT, sumT Double, cntF INT, sumF Double);
INSERT OVERWRITE TABLE q4
SELECT GP2.feat, GP1.cnT, GP1.smT, GP2.cnF, GP2.smF FROM
(select COUNT(G1.o4) cnT, SUM(regexp_extract(G1.o4,'"([0-9.]*)"(.*)',1)) smT
FROM (SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3,
SJ2.o2 o4 FROM ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(SELECT sub s1, 'bsbm:ProductType1' o1 FROM typeProductType1) T1 JOIN
(SELECT sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1))) G1 ) GP1 JOIN (
select G2.o3 feat, COUNT(G2.o5) cnF,
SUM(regexp_extract(G2.o5,'"([0-9.]*)"(.*)',1)) smF
FROM( SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4,
SJ2.o2 o5 FROM( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM
(select sub s1, 'bsbm:ProductType9' o1 FROM typeProductType9) T1 JOIN (
select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) JOIN
(select sub s1, obj o1 FROM productFeature)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2 ON
(TRIM(SJ1.s1)=TRIM(SJ2.o1))) G2 GROUP BY G2.o3 ) GP2;
```

>> Hive MQO script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String, o5 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4, SJ2.o2 o5 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM
(select sub s1, 'bsbm:ProductType9' o1 FROM typeProductType9) T1 JOIN
(select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) LEFT OUTER JOIN
(select sub s1, obj o1 FROM productFeature)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM
```

```
(select sub s1, obj o1 FROM product) T1 JOIN
(select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1));

CREATE TABLE q4 (feat String, cntT INT, sumT Double, cntF INT, sumF Double);
INSERT OVERWRITE TABLE q4
select GP2.ft, GP2.cnF, GP2.smF, GP1.cnT, GP1.smT FROM
(select COUNT(G1.price) cnT, SUM(regexp_extract(G1.price,'"([0-9.]*)"(.*)',1)) smT
FROM (select distinct o1, o2, o4, o5 price from L1) G1 ) GP1 JOIN
(select G2.ft ft, COUNT(G2.pc) cnF,
SUM(regexp_extract(G2.pc,'"([0-9.]*)"(.*)',1)) smF
FROM( select o1, o2, o3 ft, o4, o5 pc from L1 where o5 is not NULL) G2
GROUP by G2.feature) GP2;
```

SPARQL Query MG3.

```
SELECT ?feature ?country ?sumF ?countF ?sumTotal ?countTotal {
{ SELECT ?feature ?country (COUNT(?price2) As ?countF) (SUM(?price2) As ?sumF)
  { ?prod2 rdf:type bsbm-inst:ProductType1;
           rdfs:label ?lab2; bsbm:productFeature ?feature .
    ?offer2 bsbm:product ?prod2; bsbm:price ?price2; bsbm:vendor ?vend2 .
    ?vend2 bsbm:country ?country .
  } GROUP BY ?feature ?country
}
{ SELECT ?country (COUNT(?price) As ?countTotal) (SUM(?price) As ?sumTotal)
   { ?prod rdf:type bsbm-inst:ProductType1; rdfs:label ?lab1 .
    ?offer bsbm:product ?prod; bsbm:price ?price; bsbm:vendor ?vend .
    ?vend bsbm:country ?country .
  } GROUP BY ?country
} }
```

>> Hive (Naive) script.

```
CREATE TABLE q5(ft String, cn String, cnT String, smT String, cnF String, smF String);
INSERT OVERWRITE TABLE q5
SELECT GP2.feat, GP2.cn, GP1.cnT, GP1.smT, GP2.cnF, GP2.smF FROM (
select G1.o6 cn, COUNT(G1.o4) cnT, SUM(regexp_extract(G1.o4,'"([0-9.]*)"(.*)',1)) smT
FROM( SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3,
SJ2.o2 o4, SJ2.o3 o5, SJ3.o1 o6 FROM (
```

```
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
SELECT sub s1, 'bsbm:ProductType1' o1 FROM typeProductType1) T1 JOIN (
SELECT sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM product) T1 JOIN (
select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM vendor) T3 ON (T1.s1=T3.s1) ) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
SELECT sub s1, obj o1 FROM country) T1) SJ3 ON (TRIM(SJ2.o3)=TRIM(SJ3.s1)) ) G1
GROUP BY G1.o6) GP1 JOIN ( select G2.o3 feat, G2.o7 cn, COUNT(G2.o5) cnF,
SUM(regexp_extract(G2.o5,'"([0-9.]*)"(.*)',1)) smF
FROM( SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ2.o1 o4, SJ2.o2 o5, SJ2.o3 o6, SJ3.o1 o7 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, 'bsbm:ProductType1' o1 FROM typeProductType1) T1 JOIN (
select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM productFeature)T3 ON (T1.s1=T3.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM product) T1 JOIN (
select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM vendor)T3 ON (T1.s1=T3.s1) ) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM country)T1 ) SJ3 ON (TRIM(SJ2.o3)=TRIM(SJ3.s1)) ) G2
GROUP BY G2.o3, G2.o7 ) GP2 ON (TRIM(GP1.cn)=TRIM(GP2.cn));
```

>> Hive MQO script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String,
o5 String, o6 String, o7 String);
INSERT OVERWRITE TABLE L1 SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4,
SJ2.o2 o5, SJ2.o3 o6, SJ3.o1 o7 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, 'bsbm:ProductType1' o1 FROM typeProductType1) T1 JOIN (
select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM productFeature) T3 ON (T1.s1=T3.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
```

```
select sub s1, obj o1 FROM product) T1 JOIN (
select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM vendor) T3 ON (T1.s1=T3.s1) ) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1)) JOIN ( SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM country) T1 ) SJ3 ON (TRIM(SJ2.o3)=TRIM(SJ3.s1));

CREATE TABLE q5(ft String, cn String, cnT String, smT String, cnF String, smF String);
INSERT OVERWRITE TABLE q5
select GP2.ft, GP2.cn, GP1.cnT, GP1.smT, GP2.cnF, GP2.smF FROM
(select G1.cn cn, COUNT(G1.pc) cnT, SUM(regexp_extract(G1.pc,‘"([0-9.]*)"(.*)’,1)) smT
FROM (select distinct o1, o2, o4, o5 pc, o6, o7 cn from L1) G1
GROUP by G1.cn ) GP1 JOIN ( select G2.feat ft, G2.cn cn,
COUNT(G2.pc) cnF, SUM(regexp_extract(G2.pc,’"([0-9.]*)"(.*)’,1)) smF
FROM( select o1, o2, o3 feat, o4, o5 pc, o6, o7 cn from L1 where o5 is not NULL) G2
GROUP BY G2.ft, G2.cn)GP2 ON (TRIM(GP1.cn)=TRIM(GP2.cn));
```

SPARQL Query MG4.

```
SELECT ?feature ?country ?sumF ?countF ?sumTotal ?countTotal {
{ SELECT ?feature ?country (COUNT(?price2) As ?countF) (SUM(?price2) As ?sumF)
  { ?prod2 rdf:type bsbm-inst:ProductType9;
           rdfs:label ?lab2; bsbm:productFeature ?feature .
    ?offer2 bsbm:product ?prod2; bsbm:price ?price2; bsbm:vendor ?vend2 .
    ?vend2 bsbm:country ?country .
  } GROUP BY ?feature ?country
}
{ SELECT ?country (COUNT(?price) As ?countTotal) (SUM(?price) As ?sumTotal)
   { ?prod rdf:type bsbm-inst:ProductType9; rdfs:label ?lab1 .
    ?offer bsbm:product ?prod; bsbm:price ?price; bsbm:vendor ?vend .
    ?vend bsbm:country ?country .
  } GROUP BY ?country
} }
```

>> Hive (Naive) script.

```
CREATE TABLE q5(ft String, cn String, cnT String, smT String, cnF String, smF String);
INSERT OVERWRITE TABLE q5
SELECT GP2.feat, GP2.cn, GP1.cnT, GP1.smT, GP2.cnF, GP2.smF FROM (
select G1.o6 cn, COUNT(G1.o4) cnT, SUM(regexp_extract(G1.o4,’"([0-9.]*)"(.*)’,1)) smT
```

```
FROM( SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3,
SJ2.o2 o4, SJ2.o3 o5, SJ3.o1 o6 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
SELECT sub s1, 'bsbm:ProductType9' o1 FROM typeProductType9) T1 JOIN (
SELECT sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM product) T1 JOIN (
select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM vendor) T3 ON (T1.s1=T3.s1) ) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
SELECT sub s1, obj o1 FROM country) T1) SJ3 ON (TRIM(SJ2.o3)=TRIM(SJ3.s1)) ) G1
GROUP BY G1.o6) GP1 JOIN ( select G2.o3 feat, G2.o7 cn, COUNT(G2.o5) cnF,
SUM(regexp_extract(G2.o5,'"([0-9.]*)"(.*)',1)) smF
FROM( SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ2.o1 o4, SJ2.o2 o5, SJ2.o3 o6, SJ3.o1 o7 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, 'bsbm:ProductType9' o1 FROM typeProductType9) T1 JOIN (
select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM productFeature)T3 ON (T1.s1=T3.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM product) T1 JOIN (
select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM vendor)T3 ON (T1.s1=T3.s1) ) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM country)T1 ) SJ3 ON (TRIM(SJ2.o3)=TRIM(SJ3.s1)) ) G2
GROUP BY G2.o3, G2.o7 ) GP2 ON (TRIM(GP1.cn)=TRIM(GP2.cn));
```

>> Hive MQO script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String,
o5 String, o6 String, o7 String);
INSERT OVERWRITE TABLE L1 SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4,
SJ2.o2 o5, SJ2.o3 o6, SJ3.o1 o7 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, 'bsbm:ProductType9' o1 FROM typeProductType9) T1 JOIN (
select sub s1, obj o1 FROM label) T2 ON (T1.s1=T2.s1) JOIN (
```

```
select sub s1, obj o1 FROM productFeature) T3 ON (T1.s1=T3.s1) ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM product) T1 JOIN (
select sub s1, obj o1 FROM price) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM vendor) T3 ON (T1.s1=T3.s1) ) SJ2
ON (TRIM(SJ1.s1)=TRIM(SJ2.o1)) JOIN ( SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM country) T1 ) SJ3 ON (TRIM(SJ2.o3)=TRIM(SJ3.s1));

CREATE TABLE q5(ft String, cn String, cnT String, smT String, cnF String, smF String);
INSERT OVERWRITE TABLE q5
select GP2.ft, GP2.cn, GP1.cnT, GP1.smT, GP2.cnF, GP2.smF FROM
(select G1.cn cn, COUNT(G1.pc) cnT, SUM(regexp_extract(G1.pc,'"([0-9.]*)"(.*)',1)) smT
FROM (select distinct o1, o2, o4, o5 pc, o6, o7 cn from L1) G1
GROUP by G1.cn ) GP1 JOIN ( select G2.feat ft, G2.cn cn,
COUNT(G2.pc) cnF, SUM(regexp_extract(G2.pc,'"([0-9.]*)"(.*)',1)) smF FROM(
select o1, o2, o3 feat, o4, o5 pc, o6, o7 cn from L1 where o5 is not NULL)G2
GROUP by G2.ft, G2.cn) GP2 ON (TRIM(GP1.cn)=TRIM(GP2.cn));
```

D.2 Real-world Queries on Chem2Bio2RDF Dataset

```
PREFIX pubchem: <http://chem2bio2rdf.org/pubchem/resource/>
PREFIX drugbank: <http://chem2bio2rdf.org/drugbank/resource/>
PREFIX uniprot: <http://chem2bio2rdf.org/uniprot/resource/>
PREFIX kegg: <http://chem2bio2rdf.org/kegg/resource/>
PREFIX sider: <http://chem2bio2rdf.org/sider/resource/>
PREFIX medline: <http://chem2bio2rdf.org/medline/resource/>
```

SPARQL Query G5.

```
SELECT ?compound_cid (COUNT(?compound_cid) as ?active_assays {
   ?bioassay pubchem:CID ?compound_cid; pubchem:outcome ?activity;
             pubchem:Score ?score1; pubchem:gi ?gi .
   ?uniprot uniprot:gi ?gi; uniprot:geneSymbol ?gene .
   ?drugbank_interaction drugbank:gene ?gene; drugbank:DBID ?drugbank_drug .
   ?drugbank_drug drugbank:Generic_Name "Dexamethasone" . }
GROUP BY ?compound_cid
```

SPARQL Query G6.

```
SELECT ?compound_cid (COUNT(?compound_cid) as ?active_assays) {
   ?bioassay pubchem:CID ?compound_cid; pubchem:outcome ?activity;
             pubchem:Score ?score1; pubchem:gi ?gi .
   ?uniprot uniprot:gi ?gi .
   ?pathway kegg:protein ?uniprot; kegg:Pathway_name ?pathway_name .
   FILTER regex(?pathway_name,"MAPK signaling pathway","i") . }
GROUP BY ?compound_cid
```

SPARQL Query G7.

```
SELECT ?pathway_id (COUNT(?pathway_id) as ?count) {
   ?sider sider:side_effect ?side_effect; sider:cid ?compound .
   FILTER regex(?side_effect,"hepatomegaly","i")
   ?drug drugbank:CID ?compound .
   ?drug_target drugbank:DBID ?drug; drugbank:SwissProt_ID ?uniprot .
   ?pathway kegg:protein ?uniprot; kegg:pathwayid ?pathway_id .}
GROUP BY ?pathway_id
```

SPARQL Query G8.

```
SELECT ?compound_cid (COUNT(?compound_cid) as ?active_assays {
   ?bioassay pubchem:CID ?compound_cid; pubchem:outcome ?activity;
             pubchem:Score ?score1; pubchem:gi ?gi .
   ?uniprot uniprot:gi ?gi; uniprot:geneSymbol ?gene .
   ?drugbank_interaction drugbank:gene ?gene; drugbank:DBID ?drugbank_drug . }
GROUP BY ?compound_cid
```

SPARQL Query G9.

```
SELECT ?geneSymbol (COUNT(?geneSymbol) as ?pmidPerGene){
    ?gene uniprot:geneSymbol ?geneSymbol .
    ?pmid medline:gene ?gene; medline:side_effect ?sider .}
GROUP BY ?geneSymbol
```

SPARQL Query MG6.

```
SELECT ?compound_cid ?gene1 ?active_assaysPerGene ?active_assays {
   { SELECT ?compound_cid ?gene1 (COUNT(?compound_cid) as ?active_assaysPerGene)
      { ?bioassay1 pubchem:CID ?compound_cid; pubchem:outcome ?activity1;
                   pubchem:Score ?score1; pubchem:gi ?gi1 .
```

```
    ?uniprot1 uniprot:gi ?gi1; uniprot:geneSymbol ?gene1 .
    ?drugbank_interaction1 drugbank:gene ?gene1; drugbank:DBID ?drugbank_drug1 .
  } GROUP BY ?compound_cid ?gene1
}
{ SELECT ?compound_cid (COUNT(?compound_cid) as ?active_assays)
  { ?bioassay pubchem:CID ?compound_cid; pubchem:outcome ?activity;
              pubchem:Score ?score; pubchem:gi ?gi .
    ?uniprot uniprot:gi ?gi; uniprot:geneSymbol ?gene .
    ?drugbank_interaction drugbank:gene ?gene; drugbank:DBID ?drugbank_drug .
  } GROUP BY ?compound_cid
} }
```

>> Hive (Naive) script.

```
CREATE TABLE mg1 (cid String, gene String, perGene String, total String);
INSERT OVERWRITE TABLE mg1
SELECT GP2.cid, GP2.gene, GP2.essaysPerGene, GP1.essays FROM (
select G1.o1 cid, COUNT(G1.o1) essays FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ1.o4 o4, SJ2.o1 o5, SJ2.o2 o6, SJ3.o1 o7, SJ3.o2 o8 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3, T4.o1 o4 FROM (
select sub s1, obj o1 FROM pubchemCID) T1 JOIN (
select sub s1, obj o1 FROM pubchemOutcome) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubchemScore)T3 ON (T1.s1=T3.s1) JOIN (
select sub s1, obj o1 FROM pubchemGi)T4 ON (T1.s1=T4.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM uniprotGi) T1 JOIN (
select sub s1, obj o1 FROM uniprotGeneSymbol) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o4)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM drugbankGene) T1 JOIN (
select sub s1, obj o1 FROM drugbankDBID) T2 ON (T1.s1=T2.s1)) SJ3
ON (TRIM(SJ2.o2)=TRIM(SJ3.o1))) G1 GROUP BY G1.o1) GP1
JOIN ( select G2.o1 cid, G2.o6 gene, COUNT(G2.o1) essaysPerGene FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ1.o4 o4, SJ2.o1 o5, SJ2.o2 o6, SJ3.o1 o7, SJ3.o2 o8 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3, T4.o1 o4 FROM (
```

```
select sub s1, obj o1 FROM pubchemCID) T1 JOIN (
select sub s1, obj o1 FROM pubchemOutcome) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubchemScore)T3 ON (T1.s1=T3.s1) JOIN (
select sub s1, obj o1 FROM pubchemGi)T4 ON (T1.s1=T4.s1)) SJ1
JOIN ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM uniprotGi) T1 JOIN (
select sub s1, obj o1 FROM uniprotGeneSymbol) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o4)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM drugbankGene) T1 JOIN (
select sub s1, obj o1 FROM drugbankDBID) T2 ON (T1.s1=T2.s1)) SJ3
ON (TRIM(SJ2.o2)=TRIM(SJ3.o1))) G2 GROUP BY G2.o1, G2.o6) GP2
ON (TRIM(GP1.cid)=TRIM(GP2.cid));
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String,
o5 String, o6 String, o7 String, o8 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ1.o4 o4, SJ2.o1 o5, SJ2.o2 o6,
SJ3.o1 o7, SJ3.o2 o8 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3, T4.o1 o4 FROM (
select sub s1, obj o1 FROM pubchemCID) T1 JOIN (
select sub s1, obj o1 FROM pubchemOutcome) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubchemScore)T3 ON (T1.s1=T3.s1) JOIN (
select sub s1, obj o1 FROM pubchemGi)T4 ON (T1.s1=T4.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM uniprotGi) T1 JOIN (
select sub s1, obj o1 FROM uniprotGeneSymbol) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o4)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM drugbankGene) T1 JOIN (
select sub s1, obj o1 FROM drugbankDBID) T2 ON (T1.s1=T2.s1)) SJ3
ON (TRIM(SJ2.o2)=TRIM(SJ3.o1));

CREATE TABLE mg1 (cid String, gene String, perGene String, total String);
INSERT OVERWRITE TABLE mg1
select GP2.cid, GP2.geneSymbol, GP2.essaysPerGene, GP1.essays FROM
```

```
(select G1.cid cid, COUNT(G1.cid) essays FROM
(select o1 cid, o2, o3, o4, o5, o6, o7, o8 country from L1)G1
GROUP BY G1.cid )GP1 JOIN
(select G2.cid cid, G2.gene geneSymbol, COUNT(G2.cid) essaysPerGene FROM
(select o1 cid, o2, o3, o4, o5, o6 gene, o7, o8 from L1 where o5 is not NULL)G2
GROUP BY G2.cid, G2.gene)GP2 ON (TRIM(GP1.cid)=TRIM(GP2.cid));
```

SPARQL Query MG7.

```
SELECT ?compound_cid ?drugbank_drug1 ?assaysPerDrug ?active_assays {
   { SELECT ?compound_cid ?drugbank_drug1 (COUNT(?compound_cid) as ?assaysPerDrug)
     { ?bioassay1 pubchem:CID ?compound_cid; pubchem:outcome ?activity1;
                  pubchem:Score ?score1; pubchem:gi ?gi1 .
       ?uniprot1 uniprot:gi ?gi1; uniprot:geneSymbol ?gene1 .
       ?drugbank_interaction1 drugbank:gene ?gene1; drugbank:DBID ?drugbank_drug1 .
     } GROUP BY ?compound_cid ?drugbank_drug1
    }
    { SELECT ?compound_cid (COUNT(?compound_cid) as ?active_assays)
      { ?bioassay pubchem:CID ?compound_cid; pubchem:outcome ?activity;
                  pubchem:Score ?score; pubchem:gi ?gi .
        ?uniprot uniprot:gi ?gi; uniprot:geneSymbol ?gene .
        ?drugbank_interaction drugbank:gene ?gene; drugbank:DBID ?drugbank_drug .
      } GROUP BY ?compound_cid
    } }
```

>> Hive (Naive) script.

```
CREATE TABLE mg1 (cid String, gene String, perDrug String, total String);
INSERT OVERWRITE TABLE mg1
SELECT GP2.cid, GP2.drug, GP2.essaysPerDrug, GP1.essays FROM (
select G1.o1 cid, COUNT(G1.o1) essays FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ1.o4 o4, SJ2.o1 o5, SJ2.o2 o6, SJ3.o1 o7, SJ3.o2 o8 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3, T4.o1 o4 FROM (
select sub s1, obj o1 FROM pubchemCID) T1 JOIN (
select sub s1, obj o1 FROM pubchemOutcome) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubchemScore)T3 ON (T1.s1=T3.s1) JOIN (
select sub s1, obj o1 FROM pubchemGi)T4 ON (T1.s1=T4.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
```

```
select sub s1, obj o1 FROM uniprotGi) T1 JOIN (
select sub s1, obj o1 FROM uniprotGeneSymbol) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o4)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM drugbankGene) T1 JOIN (
select sub s1, obj o1 FROM drugbankDBID) T2 ON (T1.s1=T2.s1)) SJ3
ON (TRIM(SJ2.o2)=TRIM(SJ3.o1))) G1 GROUP BY G1.o1) GP1 JOIN (
select G2.o1 ccid, G2.o8 drug, COUNT(G2.o1) essaysPerDrug FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ1.o4 o4, SJ2.o1 o5, SJ2.o2 o6, SJ3.o1 o7, SJ3.o2 o8 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3, T4.o1 o4 FROM (
select sub s1, obj o1 FROM pubchemCID) T1 JOIN (
select sub s1, obj o1 FROM pubchemOutcome) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubchemScore)T3 ON (T1.s1=T3.s1) JOIN (
select sub s1, obj o1 FROM pubchemGi)T4 ON (T1.s1=T4.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM uniprotGi) T1 JOIN (
select sub s1, obj o1 FROM uniprotGeneSymbol) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o4)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM drugbankGene) T1 JOIN (
select sub s1, obj o1 FROM drugbankDBID) T2 ON (T1.s1=T2.s1)) SJ3
ON (TRIM(SJ2.o2)=TRIM(SJ3.o1))) G2 GROUP BY G2.o1, G2.o8 ) GP2
ON (TRIM(GP1.cid)=TRIM(GP2.cid));
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String, o5 String,
o6 String, o7 String, o8 String);
INSERT OVERWRITE TABLE L1 SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ1.o4 o4, SJ2.o1 o5, SJ2.o2 o6, SJ3.o1 o7, SJ3.o2 o8 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3, T4.o1 o4 FROM (
select sub s1, obj o1 FROM pubchemCID) T1 JOIN (
select sub s1, obj o1 FROM pubchemOutcome) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubchemScore)T3 ON (T1.s1=T3.s1) JOIN (
select sub s1, obj o1 FROM pubchemGi)T4 ON (T1.s1=T4.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM uniprotGi) T1 JOIN (
```

```
select sub s1, obj o1 FROM uniprotGeneSymbol) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o4)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM drugbankGene) T1 JOIN (
select sub s1, obj o1 FROM drugbankDBID) T2 ON (T1.s1=T2.s1)) SJ3
ON (TRIM(SJ2.o2)=TRIM(SJ3.o1));

CREATE TABLE mg1 (cid String, drug String, perDrug String, total String);
INSERT OVERWRITE TABLE mg1
select GP2.cid, GP2.drugid, GP2.perDrug, GP1.allEssays FROM
(select G1.cid cid, COUNT(G1.cid) allEssays FROM
(select o1 cid, o2, o3, o4, o5, o6, o7, o8 country from L1) G1
GROUP BY G1.cid ) GP1 JOIN
(select G2.cid cid, G2.drug drugid, COUNT(G2.cid) perDrug FROM
(select o1 cid, o2, o3, o4, o5, o6, o7, o8 drug from L1 where o5 is not NULL)G2
GROUP BY G2.cid, G2.drug)GP2 ON (TRIM(GP1.cid)=TRIM(GP2.cid));
```

SPARQL Query MG8.

```
SELECT ?compound_cid ?gene1 ?active_assaysPerGene ?TotalActive_assays {
   { SELECT ?compound_cid ?gene1 (COUNT(?compound_cid) as ?active_assaysPerGene)
     { ?bioassay1 pubchem:CID ?compound_cid; pubchem:outcome ?activity1;
                  pubchem:Score ?score1; pubchem:gi ?gi1 .
       ?uniprot1 uniprot:gi ?gi1; uniprot:geneSymbol ?gene1 .
       ?drugbank_interaction1 drugbank:gene ?gene1; drugbank:DBID ?drugbank_drug1 .
     } GROUP BY ?compound_cid ?gene1
    }
    { SELECT (COUNT(?compound_cid) as ?TotalActive_assays)
      { ?bioassay pubchem:CID ?compound_cid; pubchem:outcome ?activity;
                  pubchem:Score ?score; pubchem:gi ?gi .
        ?uniprot uniprot:gi ?gi; uniprot:geneSymbol ?gene .
        ?drugbank_interaction drugbank:gene ?gene; drugbank:DBID ?drugbank_drug .
      }
    } }
```

>> Hive (Naive) script.

```
CREATE TABLE mg1 (cid String, gene String, perGene String, total String);
INSERT OVERWRITE TABLE mg1
```

```
SELECT GP2.cid, GP2.gene, GP2.perGene, GP1.allEssays FROM (
select COUNT(G1.o1) allEssays FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ1.o4 o4, SJ2.o1 o5, SJ2.o2 o6, SJ3.o1 o7, SJ3.o2 o8 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3, T4.o1 o4 FROM (
select sub s1, obj o1 FROM pubchemCID) T1 JOIN (
select sub s1, obj o1 FROM pubchemOutcome) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubchemScore)T3 ON (T1.s1=T3.s1) JOIN (
select sub s1, obj o1 FROM pubchemGi)T4 ON (T1.s1=T4.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM uniprotGi) T1 JOIN (
select sub s1, obj o1 FROM uniprotGeneSymbol) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o4)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM drugbankGene) T1 JOIN (
select sub s1, obj o1 FROM drugbankDBID) T2 ON (T1.s1=T2.s1)) SJ3
ON (TRIM(SJ2.o2)=TRIM(SJ3.o1))) G1 ) GP1 JOIN (
select G2.o1 cid, G2.o6 gene, COUNT(G2.o1) perGene FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ3.s1 s3, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ1.o4 o4, SJ2.o1 o5, SJ2.o2 o6, SJ3.o1 o7, SJ3.o2 o8 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3, T4.o1 o4 FROM (
select sub s1, obj o1 FROM pubchemCID) T1 JOIN (
select sub s1, obj o1 FROM pubchemOutcome) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubchemScore)T3 ON (T1.s1=T3.s1) JOIN (
select sub s1, obj o1 FROM pubchemGi)T4 ON (T1.s1=T4.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM uniprotGi) T1 JOIN (
select sub s1, obj o1 FROM uniprotGeneSymbol) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o4)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM drugbankGene) T1 JOIN (
select sub s1, obj o1 FROM drugbankDBID) T2 ON (T1.s1=T2.s1)) SJ3
ON (TRIM(SJ2.o2)=TRIM(SJ3.o1))) G2
GROUP BY G2.o1, G2.o6 ) GP2;
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String,
```

```
o5 String, o6 String, o7 String, o8 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ1.o4 o4, SJ2.o1 o5, SJ2.o2 o6,
SJ3.o1 o7, SJ3.o2 o8 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3, T4.o1 o4 FROM (
select sub s1, obj o1 FROM pubchemCID) T1 JOIN (
select sub s1, obj o1 FROM pubchemOutcome) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubchemScore)T3 ON (T1.s1=T3.s1) JOIN (
select sub s1, obj o1 FROM pubchemGi)T4 ON (T1.s1=T4.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM uniprotGi) T1 JOIN (
select sub s1, obj o1 FROM uniprotGeneSymbol) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o4)=TRIM(SJ2.o1)) JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM drugbankGene) T1 JOIN (
select sub s1, obj o1 FROM drugbankDBID) T2 ON (T1.s1=T2.s1)) SJ3
ON (TRIM(SJ2.o2)=TRIM(SJ3.o1));

CREATE TABLE mg1 (cid String, gene String, perGene String, total String);
INSERT OVERWRITE TABLE mg1
select GP2.cid, GP2.geneSymbol, GP2.perGene, GP1.allEssays FROM
(select COUNT(G1.cid) allEssays FROM
(select o1 cid, o2, o3, o4, o5, o6, o7, o8 country from L1) G1 )GP1 JOIN
(select G2.cid cid, G2.gene geneSymbol, COUNT(G2.cid) perGene FROM
(select o1 cid, o2, o3, o4, o5, o6 gene, o7, o8 from L1 where o5 is not NULL) G2
GROUP BY G2.cid, G2.gene ) GP2;
```

SPARQL Query MG9.

```
SELECT ?geneSymbol ?PubPerGene ?TotalPub {
   { SELECT ?geneSymbol (COUNT(?geneSymbol) as ?PubPerGene)
      { ?gene uniprot:geneSymbol ?geneSymbol .
        ?pmid medline:gene ?gene; medline:side_effect ?sider .
      } GROUP BY ?geneSymbol
   }
   { SELECT (COUNT(?geneSymbol1) as ?TotalPub)
      { ?gene1 uniprot:geneSymbol ?geneSymbol1 .
        ?pmid1 medline:gene ?gene1; medline:side_effect ?sider1 .
```

```
    }
  } }
```

>> Hive (Naive) script.

```
CREATE TABLE mg4 (geneSymbol String, numPmPerGene String, numPm String);
INSERT OVERWRITE TABLE mg4
SELECT GP2.geneSymbol, GP2.numPmPerGene, GP1.numPm FROM (
select COUNT(G1.o3) numPm FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM medlineGene) T1 JOIN (
select sub s1, obj o1 FROM medlineSide_effect) T2 ON (T1.s1=T2.s1)) SJ1
JOIN ( SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM uniprotGeneSymbol)T1) SJ2
ON (TRIM(SJ1.o1)=TRIM(SJ2.s1))) G1 ) GP1 JOIN (
select G2.o3 geneSymbol, COUNT(G2.o3) numPmPerGene FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM medlineGene) T1 JOIN (
select sub s1, obj o1 FROM medlineSide_effect) T2 ON (T1.s1=T2.s1)) SJ1
JOIN ( SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM uniprotGeneSymbol)T1) SJ2
ON (TRIM(SJ1.o1)=TRIM(SJ2.s1))) G2 GROUP BY G2.o3 ) GP2;
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM medlineGene) T1 JOIN (
select sub s1, obj o1 FROM medlineSide_effect) T2
ON (T1.s1=T2.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM uniprotGeneSymbol)T1) SJ2
ON (TRIM(SJ1.o1)=TRIM(SJ2.s1));

CREATE TABLE mg4 (geneSymbol String, numPmPerGene String, numPm String);
```

```
INSERT OVERWRITE TABLE mg4
select GP2.gene, GP2.numPmPerGene, GP1.numPm FROM
(select COUNT(G1.geneSymbol) numPm FROM
(select o1 geneSymbol, o2, o3 from L1)G1 )GP1 JOIN
(select G2.geneSymbol gene,COUNT(G2.geneSymbol) numPmPerGene FROM
(select o1 geneSymbol, o2, o3 from L1) G2 GROUP BY G2.geneSymbol ) GP2;
```

SPARQL Query MG10.

```
SELECT ?disease ?geneSymbol ?PubPerDiseaseGene ?PubPerGene {
   { SELECT ?geneSymbol (COUNT(?geneSymbol) as ?PubPerDiseaseGene)
      { ?gene uniprot:geneSymbol ?geneSymbol .
        ?pmid medline:gene ?gene; medline:disease ?disease; medline:side_effect ?sid .
      } GROUP BY ?disease ?geneSymbol
   }
   { SELECT ?geneSymbol (COUNT(?geneSymbol) as ?PubPerGene)
      { ?gene1 uniprot:geneSymbol ?geneSymbol .
        ?pmid1 medline:gene ?gene1; medline:side_effect ?sider1 .
      } GROUP BY ?geneSymbol
   } }
```

>> Hive (Naive) script.

```
CREATE TABLE mg5 (disease String, gene String, pmPerGene String, numPm String);
INSERT OVERWRITE TABLE mg5
SELECT GP1.disease, GP1.geneSymbol, GP1.numPmPerGene, GP2.numPm FROM (
select G1.o3 disease, G1.o4 geneSymbol, COUNT(G1.o4) numPmPerGene FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM medlineGene) T1 JOIN (
select sub s1, obj o1 FROM medlineSide_effect) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM medlineDisease) T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM uniprotGeneSymbol)T1) SJ2
ON (TRIM(SJ1.o1)=TRIM(SJ2.s1))) G1 GROUP BY G1.o3, G1.o4) GP1 JOIN (
select G2.o3 geneSymbol, COUNT(G2.o3) numPm FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM medlineGene) T1 JOIN (
```

```
select sub s1, obj o1 FROM medlineSide_effect) T2 ON (T1.s1=T2.s1)) SJ1
JOIN ( SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM uniprotGeneSymbol)T1) SJ2
ON (TRIM(SJ1.o1)=TRIM(SJ2.s1))) G2 GROUP BY G2.o3 ) GP2
ON (TRIM(GP1.geneSymbol)=TRIM(GP2.geneSymbol));
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String, o5 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4, SJ1.s1 o5 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM medlineGene) T1 JOIN (
select sub s1, obj o1 FROM medlineSide_effect) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM medlineDisease) T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM uniprotGeneSymbol)T1) SJ2
ON (TRIM(SJ1.o1)=TRIM(SJ2.s1));

CREATE TABLE mg5(disease String, gene String, perDisGene String, perGene String);
INSERT OVERWRITE TABLE mg5
select GP2.dis, GP2.gene, GP2.pmPerDiseaseGene, GP1.pmPerGene FROM
(select G1.geneSymol gene, COUNT(G1.geneSymol) pmPerGene FROM
(select distinct o1, o2, o4 geneSymol, o5 from L1)G1
GROUP BY G1.geneSymol) GP1 JOIN
(select G2.disease dis, G2.geneSymol gene, COUNT(G2.geneSymol) pmPerDiseaseGene FROM
(select o1, o2, o3 disease, o4 geneSymol, o5 from L1 where o3 is not NULL ) G2
GROUP BY G2.disease, G2.geneSymol) GP2 ON (TRIM(GP1.gene)=TRIM(GP2.gene));
```

D.3 Real-world Queries on PubMed Dataset

```
PREFIX pubmed: <http://bio2rdf.org/pubmed_vocabulary:>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

SPARQL Query MG11.

```
SELECT ?country ?perCounty ?allCountries {
   { SELECT ?country (COUNT(?g) as ?perCounty)
```

```
    { ?pub pubmed:journal ?j; pubmed:grant ?g .
      ?g pubmed:grant_agency ?ga; pubmed:grant_country ?country .
    } GROUP BY ?country
  }
  { SELECT (COUNT(?g1) as ?allCountries)
    { ?pub1 pubmed:journal ?j1; pubmed:grant ?g1 .
      ?g1 pubmed:grant_agency ?ga1 .
    }
  } }
```

>> Hive (Naive) script.

```
CREATE TABLE p1 (country String, numPerCountry String, allCountries String);
INSERT OVERWRITE TABLE p1
SELECT GP1.country, GP1.perCountry, GP2.allCountries FROM (
select G1.o4 country, COUNT(G1.o2) perCountry FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4
FROM ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM journal) T1 JOIN (
select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1))) G1 GROUP BY G1.o4 ) GP1 JOIN (
select COUNT(G2.o2) allCountries FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM journal) T1 JOIN (
select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM grantAgency)T1 ) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1))) G2 ) GP2;
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
```

```
select sub s1, obj o1 FROM journal) T1 JOIN (
select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1));

CREATE TABLE q1 (country String, numPerCountry String, allCountries String);
INSERT OVERWRITE TABLE q1
select GP2.cntry, GP2.numPerCountry, GP1.allCountries FROM
(select COUNT(G1.grant) allCountries FROM
(select distinct o1, o2 grant, o3 from L1)G1)GP1 JOIN
(select G2.country cntry, COUNT(G2.grant) numPerCountry FROM
(select o1, o2 grant, o3, o4 country from L1 where o4 is not NULL)G2
group by G2.country)GP2;
```

SPARQL Query MG12.

```
SELECT ?country ?j ?perCounty ?perCountryPubType {
   { SELECT ?country ?j (COUNT(?g) as ?perCountyPubType)
      { ?pub pubmed:publication_type ?j; pubmed:grant ?g .
        ?g pubmed:grant_agency ?ga; pubmed:grant_country ?country .
      } GROUP BY ?country ?j
   }
   { SELECT ?country (count(?g1) as ?perCountry)
      { ?pub1 pubmed:grant ?g1 .
        ?g1 pubmed:grant_agency ?ga1; pubmed:grant_country ?country .
      } GROUP BY ?country
   } }
```

>> Hive (Naive) script.

```
CREATE TABLE p1(cn String, pType String, perCntryPubType String, perCntry String);
INSERT OVERWRITE TABLE p1
SELECT GP1.country, GP1.pubType, GP1.perCountryPubType, GP2.perCountry FROM (
select G1.o4 country, G1.o1 pubType, COUNT(G1.o2) perCountryPubType FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM pubType) T1 JOIN (
```

```
select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1))) G1 GROUP BY G1.o4, G1.o1 ) GP1
JOIN ( select G2.o3 country, COUNT(G2.o1) perCountry FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ2.o1 o2, SJ2.o2 o3 FROM (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM grant) T1 ) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o1)=TRIM(SJ2.s1))) G2 GROUP BY G2.o3) GP2
on (TRIM(GP1.country)=TRIM(GP2.country));
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM pubType) T1 LEFT OUTER JOIN (
select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1));

CREATE TABLE q2(cn String, pType String, perCntryPubType String, perCntry String);
INSERT OVERWRITE TABLE q2
select GP2.pType, GP2.cn, GP2.perCntryPubType, GP1.perCountry FROM
(select G1.country cn, COUNT(grnt) perCountry FROM
(select distinct o2 grnt, o3, o4 country from L1) G1 GROUP BY G1.country ) GP1
JOIN (select G2.pubType pType, G2.country cn, COUNT(grant) perCntryPubType FROM
(select o1 pubType, o2 grant, o3, o4 country from L1 where o1 is not NULL)G2
GROUP BY G2.pubType, G2.country ) GP2 on (TRIM(GP1.cn)=TRIM(GP2.cn));
```

SPARQL Query MG13.

```
SELECT ?author ?ptype ?perPubType ?perAuthorPubType {
   { SELECT ?author ?ptype (count(?mesh) as ?perAuthorPubType)
      { ?pub pubmed:publication_type ?ptype; pubmed:mesh_heading ?mesh;
            pubmed:author ?author .
        ?author pubmed:last_name ?ln .
      } GROUP BY ?author ?ptype
   }
   { SELECT ?ptype (count(?mesh1) as ?perPubType)
      { ?pub1 pubmed:publication_type ?ptype; pubmed:mesh_heading ?mesh1;
             pubmed:author ?author1 .
        ?author1 pubmed:last_name ?ln1.
      } GROUP BY ?ptype
   } }
```

>> Hive (Naive) script.

```
CREATE TABLE q3(auth String, pType String, perAuthPType String, perPType String);
INSERT OVERWRITE TABLE q3
SELECT GP1.author, GP1.pubType, GP1.perAuthorPubType, GP2.perPubType FROM (
select G1.o3 author, G1.o1 pubType, COUNT(G1.o2) perAuthorPubType FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType) T1 JOIN (
select sub s1, obj o1 FROM meshHeading) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM lastName)T1) SJ2
ON (TRIM(SJ1.o3)=TRIM(SJ2.s1))) G1 GROUP BY G1.o3, G1.o1 ) GP1
JOIN ( select G2.o1 pubType, COUNT(G2.o2) perPubType FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ2.o1 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType) T1 JOIN (
select sub s1, obj o1 FROM meshHeading) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM lastName)T1) SJ2
ON (TRIM(SJ1.o3)=TRIM(SJ2.s1))) G2 GROUP BY G2.o1 ) GP2
```

```
on TRIM(GP1.pubType)=TRIM(GP2.pubType);
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType) T1 JOIN (
select sub s1, obj o1 FROM meshHeading) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM lastName)T1) SJ2
ON (TRIM(SJ1.o3)=TRIM(SJ2.s1));

CREATE TABLE q3(auth String, pType String, perAuthPType String, perType String);
INSERT OVERWRITE TABLE q3
select GP1.auth, GP1.pt, GP1.perAuthPubType, GP2.perPubType FROM
(select G1.pubType pt, G1.author auth, COUNT(G1.chem) perAuthPubType FROM
(select o1 pubType, o2 chem, o3 author, o4 from L1) G1
GROUP BY G1.pubType, G1.author ) GP1
JOIN (select G2.pubType pt2, COUNT(G2.chem) perPubType FROM
(select o1 pubType, o2 chem, o3 author, o4 from L1)G2 GROUP BY G2.pubType ) GP2
on (TRIM(GP1.pt)=TRIM(GP2.pt2));
```

SPARQL Query MG14.

```
SELECT ?author ?ptype ?perPubType ?perAuthorPubType {
   { SELECT ?author ?ptype (count(?chem) as ?perAuthorPubType)
      { ?pub pubmed:publication_type ?ptype; pubmed:chemical ?chem;
             pubmed:author ?author .
        ?author pubmed:last_name ?ln .
      } GROUP BY ?author ?ptype
   }
   { SELECT ?ptype (count(?chem1) as ?perPubType)
      { ?pub1 pubmed:publication_type ?ptype; pubmed:chemical ?chem1;
              pubmed:author ?author1 .
        ?author1 pubmed:last_name ?ln1.
      } GROUP BY ?ptype
```

} }

>> Hive (Naive) script.

```
CREATE TABLE q4 (auth String, pType String, perAuthPType String, perType String);
INSERT OVERWRITE TABLE q4
SELECT GP1.author, GP1.pubType, GP1.perAuthorPubType, GP2.perPubType FROM (
select G1.o3 author, G1.o1 pubType, COUNT(G1.o2) perAuthorPubType FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType) T1 JOIN (
select sub s1, obj o1 FROM chemical) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM lastName)T1) SJ2
ON (TRIM(SJ1.o3)=TRIM(SJ2.s1))) G1
GROUP BY G1.o3, G1.o1 ) GP1 JOIN (
select G2.o1 pubType, COUNT(G2.o2) perPubType FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3,
SJ2.o1 o4 FROM ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType) T1 JOIN (
select sub s1, obj o1 FROM chemical) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM lastName)T1) SJ2
ON (TRIM(SJ1.o3)=TRIM(SJ2.s1))) G2 GROUP BY G2.o1 ) GP2
on TRIM(GP1.pubType)=TRIM(GP2.pubType);
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType) T1 JOIN (
select sub s1, obj o1 FROM chemical) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM lastName)T1) SJ2
```

```
ON (TRIM(SJ1.o3)=TRIM(SJ2.s1));

CREATE TABLE q4(auth String, pType String, perAuthPType String, perType String);
INSERT OVERWRITE TABLE q4
select GP1.auth, GP1.pt, GP1.perAuthPubType, GP2.perPubType FROM
(select G1.pubType pt, G1.author auth, COUNT(G1.chem) perAuthPubType FROM
(select o1 pubType, o2 chem, o3 author, o4 from L1) G1
GROUP BY G1.pubType, G1.author ) GP1 JOIN
(select G2.pubType pt2, COUNT(G2.chem) perPubType FROM
(select o1 pubType, o2 chem, o3 author, o4 from L1)G2
GROUP BY G2.pubType ) GP2 on (TRIM(GP1.pt)=TRIM(GP2.pt2));
```

SPARQL Query MG15. For publication type Journal Article.

```
SELECT ?ln ?perAuthor ?allAuthors {
   { SELECT ?ln (count(?chem) as ?perAuthor)
      { ?pub pubmed:publication_type ‘‘Journal Article’’; pubmed:chemical ?chem;
            pubmed:author ?author .
        ?author pubmed:last_name ?ln .
      } GROUP BY ?ln
   }
   { SELECT (count(?chem1) as ?allAuthors)
      { ?pub1 pubmed:publication_type ‘‘Journal Article’’; pubmed:chemical ?chem1;
             pubmed:author ?author1 .
        ?author1 pubmed:last_name ?ln1.
      }
   } }
```

>> Hive (Naive) script.

```
CREATE TABLE q5 (author String, perAuth String, allAuth String);
INSERT OVERWRITE TABLE q5
SELECT GP1.authorLastname, GP1.perAuthor, GP2.allAuthors FROM (
select G1.o4 authorLastName, COUNT(G1.o2) perAuthor FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4
FROM ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType where obj = ‘ "Journal Article" .’) T1
JOIN ( select sub s1, obj o1 FROM chemical) T2 ON (T1.s1=T2.s1) JOIN (
```

```
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM lastName)T1) SJ2
ON (TRIM(SJ1.o3)=TRIM(SJ2.s1))) G1 GROUP BY G1.o4) GP1 JOIN (
select COUNT(G2.o2) allAuthors FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4
FROM ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType where obj = ‘ "Journal Article" .’) T1
JOIN ( select sub s1, obj o1 FROM chemical) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM lastName) T1) SJ2
ON (TRIM(SJ1.o3)=TRIM(SJ2.s1))) G2 ) GP2;
```

>> **Hive (MQO) script.**

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o3, SJ2.o1 o4 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType where obj = ‘ "Journal Article" .’) T1
JOIN ( select sub s1, obj o1 FROM chemical) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM lastName)T1) SJ2
ON (TRIM(SJ1.o3)=TRIM(SJ2.s1));

CREATE TABLE q4 (author String, perAuth String, allAuth String);
INSERT OVERWRITE TABLE q4
select GP1.auth, GP1.perAuthor, GP2.allAuthors FROM
(select G1.authorLastName auth, COUNT(G1.chem) perAuthor FROM
(select o1, o2 chem, o3, o4 authorLastName from L1) G1
GROUP BY G1.authorLastName ) GP1 JOIN
(select COUNT(G2.chem) allAuthors FROM
(select o1, o2 chem, o3, o4 from L1 ) G2 ) GP2;
```

SPARQL Query MG16. For publication type News.

```
SELECT ?ln ?perAuthor ?allAuthors {
   { SELECT ?ln (count(?chem) as ?perAuthor)
```

```
    { ?pub pubmed:publication_type ‘‘News’’; pubmed:chemical ?chem;
          pubmed:author ?author .
      ?author pubmed:last_name ?ln .
    } GROUP BY ?ln
  }
  { SELECT (count(?chem1) as ?allAuthors)
    { ?pub1 pubmed:publication_type ‘‘News’’; pubmed:chemical ?chem1;
          pubmed:author ?author1 .
      ?author1 pubmed:last_name ?ln1.
    }
  } }
```

SPARQL Query MG17.

```
SELECT ?country ?perCounty ?allCountries {
  { SELECT ?country (count(?g) as ?perCounty)
    { ?pub pubmed:publication_type ‘‘Journal Article’’; pubmed:journal ?j;
          pubmed:grant ?g .
      ?g pubmed:grant_agency ?ga; pubmed:grant_country ?country .
    } GROUP BY ?country
  }
  { SELECT (count(?g1) as ?allCountries)
    { ?pub1 pubmed:publication_type ‘‘Journal Article’’; pubmed:journal ?j1;
          pubmed:grant ?g1 .
      ?g pubmed:grant_agency ?ga1 .
    }
  } }
```

>> Hive (Naive) script.

```
CREATE TABLE p1 (country String, numPerCountry String, allCountries String);
INSERT OVERWRITE TABLE p1
SELECT GP1.country, GP1.perCountry, GP2.allCountries FROM (
select G1.o4 country, COUNT(G1.o2) perCountry FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o5, SJ2.o1 o3,
SJ2.o2 o4 FROM ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2,T3.o1 o3 FROM (
select sub s1, obj o1 FROM journal) T1 JOIN (
select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubType where obj = ‘ "Journal Article" .’)T3
```

```
ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1))) G1 GROUP BY G1.o4
) GP1 JOIN ( select COUNT(G2.o2) allCountries FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ1.o3 o4,
SJ2.o1 o3 FROM ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM journal) T1 JOIN (
select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubType where obj = ‘ "Journal Article" .’)T3
ON (T1.s1=T3.s1)) SJ1 JOIN ( SELECT T1.s1 s1, T1.o1 o1 FROM (
select sub s1, obj o1 FROM grantAgency)T1 ) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1))) G2 ) GP2;
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String, o5 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4, SJ1.o3 o5 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM journal) T1 JOIN (
select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM pubType where obj = ‘ "Journal Article" .’)T3
ON (T1.s1=T3.s1)) SJ1 JOIN ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1));

CREATE TABLE q1 (cn String, numPerCntry String, allCountries String);
INSERT OVERWRITE TABLE q1
select GP2.cntry, GP2.numPerCountry, GP1.allCountries FROM
(select COUNT(G1.grant) allCountries FROM
(select distinct o1, o2 grant, o3, o5 from L1)G1)GP1 JOIN
(select G2.country cntry, COUNT(G2.grant) numPerCountry FROM
(select o1, o2 grant, o3, o4 country, o5 from L1 where o4 is not NULL)G2
group by G2.country)GP2;
```

SPARQL Query MG18.

```
SELECT ?country ?author ?perCounty ?perAuthorCountry {
   { SELECT ?country ?author (count(?g) as ?perCountyPubType)
      { ?pub pubmed:publication_type ‘‘Journal Article’’; pubmed:author ?author;
            pubmed:grant ?g .
        ?g pubmed:grant_agency ?ga; pubmed:grant_country ?country .
      } GROUP BY ?country ?author
   }
   { SELECT ?country (count(?g1) as ?perCountry)
      { ?pub1 pubmed:publication_type ‘‘Journal Article’’; pubmed:grant ?g1 .
        ?g1 pubmed:grant_agency ?ga1; pubmed:grant_country ?country .
      } GROUP BY ?country
   } }
```

>> Hive (Naive) script.

```
CREATE TABLE p1 (cn String, auth String, perCntryAuth String, perCntry String);
INSERT OVERWRITE TABLE p1
SELECT GP1.country, GP1.author, GP1.perCountryAuthor, GP2.perCountry FROM (
select G1.o4 country, G1.o5 author, COUNT(G1.o2) perCountryAuthor FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4,
SJ1.o3 o5 FROM ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType where obj = ‘ "Journal Article" .’) T1
JOIN ( select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1) JOIN (
select sub s1, obj o1 FROM author)T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1))) G1 GROUP BY G1.o4, G1.o5
) GP1 JOIN ( select G2.o3 country, COUNT(G2.o1) perCountry FROM (
SELECT SJ1.s1 s1, SJ2.s1 s2, SJ1.o1 o1, SJ2.o1 o2, SJ2.o2 o3,
SJ1.o2 o4 FROM ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grant) T1 JOIN (
select sub s1, obj o1 FROM pubType where obj = ‘ "Journal Article" .’)T2
ON (T1.s1=T2.s1)) SJ1 JOIN ( SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o1)=TRIM(SJ2.s1))) G2 GROUP BY G2.o3 ) GP2
```

```
on (TRIM(GP1.country)=TRIM(GP2.country));
```

>> Hive (MQO) script.

```
CREATE TABLE L1(o1 String, o2 String, o3 String, o4 String, o5 String);
INSERT OVERWRITE TABLE L1
SELECT SJ1.o1 o1, SJ1.o2 o2, SJ2.o1 o3, SJ2.o2 o4, SJ1.o3 o5 FROM (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2, T3.o1 o3 FROM (
select sub s1, obj o1 FROM pubType where obj = ‘ "Journal Article" .’) T1
JOIN ( select sub s1, obj o1 FROM grant) T2 ON (T1.s1=T2.s1) LEFT OUTER JOIN
( select sub s1, obj o1 FROM author) T3 ON (T1.s1=T3.s1)) SJ1 JOIN (
SELECT T1.s1 s1, T1.o1 o1, T2.o1 o2 FROM (
select sub s1, obj o1 FROM grantAgency)T1 JOIN (
select sub s1, obj o1 FROM grantCountry) T2 ON (T1.s1=T2.s1)) SJ2
ON (TRIM(SJ1.o2)=TRIM(SJ2.s1));

CREATE TABLE q2(auth String, cn String, perCntryAuth String, perCntry String);
INSERT OVERWRITE TABLE q2
select GP2.auth, GP2.cntry, GP2.perCountryAuthor, GP1.perCountry FROM
(select G1.cn cntry, COUNT(grnt) perCountry FROM
(select distinct o1, o2 grnt, o3, o4 cn from L1) G1 GROUP BY G1.cn ) GP JOIN
(select G2.author auth, G2.country cntry, COUNT(grant) perCountryAuthor FROM
(select o1, o2 grant, o3, o4 country, o5 author from L1 where o5 is not NULL)G2
GROUP BY G2.author, G2.country ) GP2
on (TRIM(GP1.cntry)=TRIM(GP2.cntry));
```