A SCALABLE ANALYSIS FRAMEWORK FOR LARGE-SCALE RDF DATA

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Long Cheng

Department of Electronic Engineering

National University of Ireland Maynooth

This dissertation is submitted for the degree of

Doctor of Philosophy

Supervisors: Dr. Tomas Ward

Prof. Georgios Theodoropoulos (external)

Dr. Spyros Kotoulas (external)

2014

Declaration

I hereby declare that this thesis is my own work and has not been submitted in any form for another award at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Long Cheng
2014

Acknowledgements

Foremost, I would like to express my sincerest thanks to my supervisors Dr. Tomas Ward, Prof. Georgios Theodoropoulos and Dr. Spyros Kotoulas, for their constant guidance and support throughout the years of my PhD work. In the meantime, I am also greatly appreciated of their kindness, patience and encouragement that let me feel more confident on research and grow gradually as an independent research scientist.

I would like to thank all the staff members in the Department of Electronic Engineering and the Department of Computer Science at NUI Maynooth, for the great time I had in our group. I enjoyed the atmosphere and also their friendship. Since this research work was sponsored by Irish Research Council and co-funded by IBM, their support is gratefully acknowledged.

Finally, I would like to express my gratitude to my family, for their persistent support, not only during this work, but also throughout my life. I would also give my special thanks to my fiancee - Yang Liu, for her constant support and understanding in the past years and enjoying life together with me in the future.

List of Publications

Published

Robust and Efficient Large-large Table Outer Joins on Distributed Infrastructures
Long Cheng, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.
In *Euro-Par '14: Proc. 20th International European Conference on Parallel Processing*, pages 258-269, Porto, Portugal, 2014.

Efficiently Handling Skew in Outer Joins on Distributed Systems
Long Cheng, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.
In *CCGrid '14: Proc. 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 295-304, Chicago, USA, 2014.

QbDJ: A Novel Framework for Handling Skew in Parallel Join Processing on Distributed Memory
Long Cheng, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.
In *HPCC '13: Proc. 15th IEEE International Conference on High Performance Computing and Communications*, pages 1519-1527, Zhangjiajie, China, 2013.

Runtime Characterisation of Triple Stores
Long Cheng, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.
In *CSE '12: Proc. 15th IEEE International Conference on Computational Science and Engineering*, pages 66-73, Paphos, Cyprus, 2012.

Runtime Characterisation of Triple Stores: An Initial Investigation
Long Cheng, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.
In *ISSC '12: Proc. 23rd IET Irish Signals and Systems Conference*, pages 1-6, Maynooth, Ireland, 2012.

Accepted

Efficient Parallel Dictionary Encoding for RDF Data
Long Cheng, Avinash Malik, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.
In *WebDB '14: Proc. of the 17th International Workshop on the Web and Databases*, Snowbird, USA, 2014.

A Two-tier Index Architecture for Fast Processing Large RDF Data over Distributed Memory
Long Cheng, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.
In *HT '14: Proc. 25th ACM International Conference on Hypertext and Social Media*, Santiago, Chile, 2014.

Robust Skew-resistant Parallel Joins in Shared-nothing Systems
Long Cheng, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.
In *CIKM '14: Proc. 23rd ACM International Conference on Information and Knowledge Management*, Shanghai, China, 2014.

Design and Evaluation of Parallel Hashing over Large-scale Data
Long Cheng, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.
In *HiPC '14: Proc. 21st IEEE International Conference on High Performance Computing*, Goa, India, 2014.

In Submission

RDF-ReHashed: Fast Distributed Loading and Querying of Large RDF Datasets.
Long Cheng, Spyros Kotoulas, Tomas Ward, Georgios Theodoropoulos.

Massively Parallel Reasoning Under the Well-founded Semantics using X10.
Ilias Tachmazidis, Long Cheng, Spyros Kotoulas, Grigoris Antoniou, Tomas Ward

Abbreviations

APGAS	Asynchronous Partitioned Global Address Space
BGP	Basic Graph Pattern
BSBM	Berlin SPARQL Benchmark
BTC	Billion Triple Challenge
CAS	Compare-and-swap
DBMS	Database Management System
DBPSB	DBpedia SPARQL Benchmark
DER	Duplication and Efficient Redistribution
DHT	Distributed Hash Table
GPU	Graphic Processing Unit
LUBM	Lehigh University Benchmark
MPI	Message Passing Interface
NUMA	Non-uniform Memory Access
OpenMP	Open Multi-Processing
OWL	Web Ontology Language
PGAS	Partitioned Global Address Space
PRPD	Partial Redistribution & Partial Duplication
PRPQ	Partial Redistribution & Partial Query
QMpH	Query Mixes per Hour
QpS	Query per Second
RDBMS	Relational Database Management System
RDF	Resource Description Framework
SAN	Storage Area Network
SIP	Sideways Information Passing
SPARQL	Simple Protocol and RDF Query Language
SQL	Structured Query Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium

Abstract

With the growth of the Semantic Web, the availability of RDF datasets from multiple domains as Linked Data has taken the corpora of this web to a terabyte-scale, and challenges modern knowledge storage and discovery techniques. Research and engineering on RDF data management systems is a very active area with many standalone systems being introduced. However, as the size of RDF data increases, such single-machine approaches meet performance bottlenecks, in terms of both data loading and querying, due to the limited parallelism inherent to symmetric multi-threaded systems and the limited available system I/O and system memory. Although several approaches for distributed RDF data processing have been proposed, along with clustered versions of more traditional approaches, their techniques are limited by the trade-off they exploit between loading complexity and query efficiency in the presence of big RDF data. This thesis then, introduces a scalable analysis framework for processing large-scale RDF data, which focuses on various techniques to reduce inter-machine communication, computation and load-imbalancing so as to achieve fast data loading and querying on distributed infrastructures.

The first part of this thesis focuses on the study of RDF store implementation and parallel hashing on big data processing. (1) A system-level investigation of RDF store implementation has been conducted on the basis of a comparative analysis of runtime characteristics of a representative set of RDF stores. The detailed time cost and system consumption is measured for data loading and querying so as to provide insight into different triple store implementation as well as an understanding of performance differences between different platforms. (2) A high-level structured parallel hashing approach over distributed memory is proposed and theoretically analyzed. The detailed performance of hashing implementations using different lock-free strategies has been characterized through extensive experiments, thereby allowing system developers to make a more informed choice for the implementation of their high-performance analytical data processing systems.

The second part of this thesis proposes three main techniques for fast processing of large RDF data within the proposed framework. (1) A very efficient parallel dictionary encoding algorithm, to avoid unnecessary disk-space consumption and reduce computational com-

plexity of query execution. The presented implementation has achieved notable speedups compared to the state-of-art method and also has achieved excellent scalability. (2) Several novel parallel join algorithms, to efficiently handle skew over large data during query processing. The approaches have achieved good load balancing and have been demonstrated to be faster than the state-of-art techniques in both theoretical and experimental comparisons. (3) A two-tier dynamic indexing approach for processing SPARQL queries has been devised which keeps loading times low and decreases or in some instances removes inter-machine data movement for subsequent queries that contain the same graph patterns. The results demonstrate that this design can load data at least an order of magnitude faster than a clustered store operating in RAM while remaining within an interactive range for query processing and even outperforms current systems for various queries.

Contents

Contents **xiii**

List of Figures **xix**

List of Tables **xxiii**

1 Introduction **1**

1.1 Introduction 1

1.1.1 RDF Data 2

1.1.2 SPARQL 4

1.1.3 RDF Stores 5

1.2 Objectives of this Thesis 8

1.3 Contributions of this Thesis 10

1.4 Outline of this Thesis 14

2 Related Work **17**

2.1 Introduction 17

2.2 High Performance RDF Data Management Systems 18

2.2.1 Sequential Solutions 18

2.2.2 Parallel Solutions 22

2.3 RDF Store Benchmarks 26

2.3.1 RDF Benchmarks 26

2.3.2 Benchmark Datasets 27

2.3.3 Evaluation Work 27

2.4 RDF Data Compression 28

2.5 Parallel Join Approaches 29

2.5.1 Inner Joins 30

2.5.2 Outer Joins 35

2.6 X10 Parallel Programming Language 38
2.7 Conclusion 39

3 Runtime Characterization of Triple Stores 41
3.1 Introduction 41
3.2 RDF Store Querying 42
3.2.1 Query Planning 43
3.2.2 Query Execution 43
3.3 Methodology and Metrics 45
3.4 Experimental Settings 46
3.4.1 Benchmark 46
3.4.2 Platform 47
3.4.3 Setup 47
3.5 Results and Discussion 48
3.5.1 Loading 48
3.5.2 QMpH 50
3.5.3 Cost Breakdown 50
3.5.4 Planning and Execution 51
3.5.5 Number of Scans and Scan Time 52
3.5.6 Number of Lookups and Read in Pages 54
3.5.7 CPU Usage 54
3.6 Conclusions 54

4 Design and Evaluation of Parallel Hashing over Large-scale Data 57
4.1 Introduction 57
4.2 Theoretical Analysis of Hashing Frameworks 61
4.2.1 Distribution 61
4.2.2 Slot Probing 62
4.2.3 Memory Contention 62
4.2.4 Performance Comparison 63
4.3 Parallel Hashing 64
4.3.1 Distribution 64
4.3.2 Processing 65
4.4 Evaluation 69
4.4.1 Comparison of Frameworks 69
4.4.2 Structured Distributed Hash Tables 71

4.4.3 Hybrid Parallel Hash Tables . . . 71
4.4.4 Impact Factors . . . 75
4.4.5 Comparison with Current Implementations . . . 76
4.5 Discussion . . . 77
4.6 Conclusions . . . 78

5 Scalable RDF Data Compression using X10 79
5.1 Introduction . . . 79
5.2 RDF Compression . . . 81
5.2.1 Main Algorithm . . . 81
5.2.2 Detailed Implementation . . . 82
5.3 Improvements . . . 85
5.3.1 I/O and Data Transfers . . . 86
5.3.2 Flexible Memory Footprint . . . 86
5.3.3 Transactional Data Processing . . . 87
5.3.4 Incremental Update . . . 87
5.3.5 Algorithmic Complexity . . . 88
5.4 Evaluation . . . 88
5.4.1 Experimental setup . . . 88
5.4.2 Runtime . . . 89
5.4.3 Scalability . . . 93
5.4.4 Load Balancing . . . 95
5.5 Discussion . . . 97
5.6 Conclusions . . . 98

6 A Novel Framework for Handling Skew in Parallel Joins on Distributed Systems 99
6.1 Introduction . . . 99
6.2 Query-based Distributed Join . . . 100
6.2.1 Framework . . . 100
6.2.2 Handling Data Skew . . . 102
6.2.3 Comparison with other Approaches . . . 103
6.3 Applying to Outer Joins . . . 104
6.4 Implementation . . . 106
6.4.1 Parallel Join Processing . . . 106
6.4.2 The PRPD-based Methods using X10 . . . 108

6.5 Evaluation of Inner Joins . . . 109
6.5.1 Platform . . . 109
6.5.2 Datasets . . . 109
6.5.3 Setup . . . 109
6.5.4 Runtime . . . 110
6.5.5 Network Communication . . . 111
6.5.6 Load Balancing . . . 111
6.5.7 Scalability . . . 112
6.6 Evaluation of Outer Joins . . . 113
6.6.1 Runtime . . . 113
6.6.2 Network Communication . . . 116
6.6.3 Load Balancing . . . 116
6.6.4 Scalability . . . 118
6.7 Conclusions . . . 118

7 High Performance Skew-Resistant Parallel Joins in Shared-Nothing Systems 119
7.1 Introduction . . . 119
7.2 PRPQ Joins . . . 120
7.2.1 The PRPQ Algorithm . . . 120
7.2.2 Compared to the QUERY-BASED Algorithm . . . 121
7.2.3 Comparison with PRPD . . . 121
7.3 Theoretical Comparison of Parallel Join Approaches . . . 122
7.3.1 Skew in Parallel Joins . . . 123
7.3.2 PRPD Joins . . . 124
7.3.3 Query-based Joins . . . 126
7.3.4 PRPQ Joins . . . 127
7.3.5 Performance Comparison . . . 128
7.4 Implementation . . . 130
7.4.1 Local Skew . . . 130
7.4.2 Parallel Processing . . . 130
7.5 Experimental Evaluation . . . 133
7.5.1 Runtime . . . 134
7.5.2 Network Communication . . . 137
7.5.3 Load Balancing . . . 138
7.5.4 Scalability . . . 139

7.5.5 Comparison with Hash-based Joins . . . 140
7.6 Conclusions . . . 141

8 Fast Distributed Loading and Querying of Large RDF Data 143
8.1 Introduction . . . 143
8.2 Data Loading . . . 145
8.3 Data Querying . . . 146
8.4 Distributed Filters . . . 151
8.5 Evaluation . . . 153
8.5.1 Setup . . . 153
8.5.2 Benchmark . . . 154
8.5.3 Data Loading Time . . . 154
8.5.4 General Query Performance . . . 155
8.5.5 Indexes and Filters . . . 157
8.5.6 Load Balancing and Scalability . . . 159
8.6 Discussion . . . 160
8.7 Conclusion . . . 162

9 Conclusions and Future Work 163
9.1 Summary of Conclusions . . . 163
9.2 Future Work . . . 166
9.3 Concluding Remarks . . . 167

References 169

Appendix A The Detailed Implementation of *Query with Counters* 179

Appendix B Rewritten LUBM SPARQL Queries 183

List of Figures

1.1 Linking Open Data cloud diagram (taken from [37]). 2
1.2 An example of RDF triples. 3
1.3 An example of RDF graph. 3
1.4 An example of query graph pattern. 5
1.5 RDF data stored as property tables. 7
1.6 RDF data stored as triples in a big table. 7
1.7 RDF data stored as vertically tables. 8
1.8 General design of our parallel framework, which includes two main parts, the data loading and data querying. This thesis focuses on the techniques used in three core parts for a system: encoding, joins and indexing. 12

2.1 An example of SPO indexing in a Hexastore [121]. 18
2.2 An example of bit-matrix structure for storing RDF data in BitMat [11]. . . 19
2.3 An example of the tree of predicate path that is used for filtering non-useful results during query executions. 21
2.4 An RDF graph and the responsible triples. 23
2.5 Two queries in the form of graph patterns. 23
2.6 The similar-size partitioning method over a two-node system. 24
2.7 The hash-based partitioning method over a two-node system. 24
2.8 The graph-based partitioning method over a two-node system. 26
2.9 The hash-based distributed join approach. The dashed square refers to the remote computation nodes and objects. 31
2.10 Duplication-based distributed join framework. 31
2.11 An example of the data movements in PRPD implementation. 33
2.12 Hash-based distributed outer joins. 36
2.13 Duplication-based distributed outer joins. 37

3.1 The work flow of the general query process in triple stores. 43
3.2 Pseudo codes of four counters in a scan implementation. 45
3.3 Data loading time on the two platforms. 49
3.4 Disk space required for various datasets. 49
3.5 QMpH for various datasets on the two platforms. 49
3.6 The planning time of Query 12 by varying the number of triples (in logscale). 52
3.7 The execution time of Query 5 by varying the number of triples (in logscale). 52
3.8 The number of index scans of Query 8 by varying the number of triples. . . 53
3.9 The scan time of Query 8 by varying the number of triples. 53
3.10 The number of triple lookups of Query 5 by varying the number of triples. For each dataset, Jena and Sesame performs nearly the same, are much smaller than RDF-3X. 55
3.11 The number of read in pages of Query 5 by varying the number of triples. . 55

4.1 Distributed-level parallelism. 58
4.2 Thread-level parallelism 58
4.3 Structured parallelism 58
4.4 CAS-based Implementation. 66
4.5 Range-based Implementation. 67
4.6 Performance comparison of three frameworks. 70
4.7 Time cost with varying number of threads for `SDHT`. 72
4.8 Time cost with varying size of input for `SDHT`. 72
4.9 Time cost with varying the number of threads for `HPHT`. 72
4.10 Runtime by varying *Zipfian factor* in each implementation. 73
4.11 Time cost with different load factors. 74
4.12 Time cost with varying the parameter i. 74

5.1 Data flow of the RDF compression in our implementation. 81
5.2 Throughput of the two implementations using 192 cores, based on disk-based and memory-based cases with the four datasets. 91
5.3 Runtime by varying nodes. 94
5.4 Speedups by varying nodes. 94
5.5 Runtime by varying size. 95

6.1 Query-based Distributed Join Framework. The dashed rectangle refers to the remote computation nodes and objects. 100

6.2 The data structure used in query-based distributed join: (a) the local hash tables of S (left), and (b) the query keys of a remote node and its corresponding returned values (right). 101
6.3 An example of the query-based implementation over a two-node system. . . 102
6.4 The Query with Counters approach for outer joins. The dashed rectangle refers to the remote computation nodes and objects. 104
6.5 The data structure used in QC algorithm: (a) the local hash tables of S (left), and (b) the query keys of a remote node and its corresponding returned values (right). 105
6.6 Runtime comparison of the three different algorithms. The join is implemented on $256M \bowtie 1B$ with different skew by using 192 cores. 110
6.7 The average number of received tuples (or keys) for each place of the three different algorithms. 111
6.8 The detailed time cost of query-based join approach on different key distributions by increasing number of cores. 113
6.9 Runtime comparison of the four algorithms under different skews (with selectivity factor 100% over 192 cores). 114
6.10 Runtime of the four algorithms under low skew by varying the join selectivity factor ($skew = 1$ over 192 cores). 115
6.11 Runtime of the four algorithms under high skew by varying the join selectivity factor ($skew = 1.4$ over 192 cores). 115
6.12 The average number of received tuples (or keys) for each place under different skews (with selectivity factor 100% over 192 cores). 117
6.13 The runtime breakdown of the QC algorithm under skews by varying number of cores (with selectivity factor 100%). 117

7.1 The PRPQ join approach. Only the high skew part of S implements the query operations, and the rest is processed as the basic hash method. 121
7.2 Distribution of the tuples in S at each node based on the rank of keys. . . . 123
7.3 Runtime of the four algorithms. 135
7.4 Runtime of PRPD and PRPQ with increasing threshold t over different datasets ($64M \bowtie 1B$ with 192 cores). 136
7.5 Average number of received tuples at each place by varying the threshold ($64M \bowtie 1B$ with 192 cores). 138

7.6 The runtime breakdown of PRPQ under different skews by increasing the cores ($64M \bowtie 1B$). 139
7.7 Speedup ration over the hash algorithm under different skews by varying the nodes ($64M \bowtie 1B$). 140

8.1 The triples and the primary index for a simple two node system (vertical tables in the dashed square compose the $P \rightarrow SO$ and $PS \rightarrow O$ part of l_1). . . 145
8.2 An example of a simple SPARQL query graph and its query plan. 146
8.3 Example of query execution and secondary index building. 149
8.4 A complex SPARQL query graph and the *join* in its graph path. 151
8.5 Runtime for RDF-3X and 4store, and detailed runtime of each implementation for our system (over Q2 and Q9 using 192 cores). 157
8.6 Number of redistributed tuples. 158
8.7 Number of elements in a filter (index). 158

List of Tables

2.1 An example of data partitioning in the PRPD algorithm 34

3.1 Metrics List 46
3.2 The Configurations of Test Platforms 47
3.3 Special queries for RDF stores with 250M triples on standard platform . . . 50
3.4 Breakdown of different queries for 250M triples on the standard platform (in %) 51

4.1 Experimental Parameters. 70
4.2 Detailed Time cost of processing different integer lengths 74
4.3 Comparison with results presented in [51] (time in seconds) 76

5.1 Dataset information and compression achieved 90
5.2 Disk-based runtime and rates of compression (192 cores) 91
5.3 In-memory runtime and rates of compression (192 cores) 91
5.4 Processing 1M statements in the transactional scenario 92
5.5 Incremental update scenario with different chunk size 93
5.6 Detailed term information during encoding 1.1 billion triples 97
5.7 Comparison of received data for each computing node when processing 1.1 billion triples using 192 cores (in millions) 97

6.1 The number of received tuples or keys (in millions) 112
6.2 The number of received tuples at each place (millions) 117

7.1 Datasets with different key distribution and partitioning used in our tests . . 134
7.2 Speedup achieved by PRPQ over PRPD with varying the size of inputs (using 192 cores). 137
7.3 Detailed number of received tuples at each place (millions) 139

8.1 Time to load 1.1 billion triples 155
8.2 Execution times for the LUBM queries over RDF-3X and 4store with cold and warm runs, as well as our system with the primary index l_1 and second-level index l_2 (ms) 156
8.3 Number of received tuples at each core (millions) for 192 cores 159
8.4 Runtime by varying the number of cores over 2nd-level index 160

Chapter 1

Introduction

1.1 Introduction

The Semantic Web [13], which is considered an extension of the current World Wide Web, is now becoming mainstream. As the information in this web is given a well-defined meaning and encoded in a machine-readable format, it possess plenty of special characteristics not available with the traditional web, such as amenability to machine processing, information lookup and knowledge inference.

This web is founded on the concept of Linked Data [15], a term used to describe the practices of exposing, sharing and connecting information on the web using recent W3C specifications such as *Resource Description Framework* [117] (RDF, details given later). Linked Data is fast becoming the dominant model for cross-database data integration. It can been seen from Figure 1.1 that there has already been large amounts of data from different domains interlinked with each other and compose a large data cloud. Up until now, this cloud has consisted of more than 200 data sources covering many well-known areas, such as general knowledge (DBpedia [12]), bioinformatics (Uniprot [10]), GIS (geoname [123], linkedgeodata [104]) and web-page annotations (RDFa [5], microformats [74]), which have contributed to more than 25 billion data items already [42]. In addition to this, it is increasingly prevalent particularly among governments and enterprises that see RDF as a more flexible way to represent their data, notably the US government (data.gov) and that of the UK (data.gov.uk) as well as Google, Bing and Yahoo (schema.org). Moreover, in tandem with the increasing availability of such data and corresponding technologies, an increasing number of software platforms now use RDF as well (e.g. the BBC website [94]).

With the rapid increase of the cloud and the increase in published data from different domains, the potential for new knowledge synthesis and discovery increases immensely.

Capitalizing on this potential requires Semantic Web applications which are capable of integrating the information available from this rapidly expanding web. The web engineering challenges which this presents are currently pushing computing boundaries.

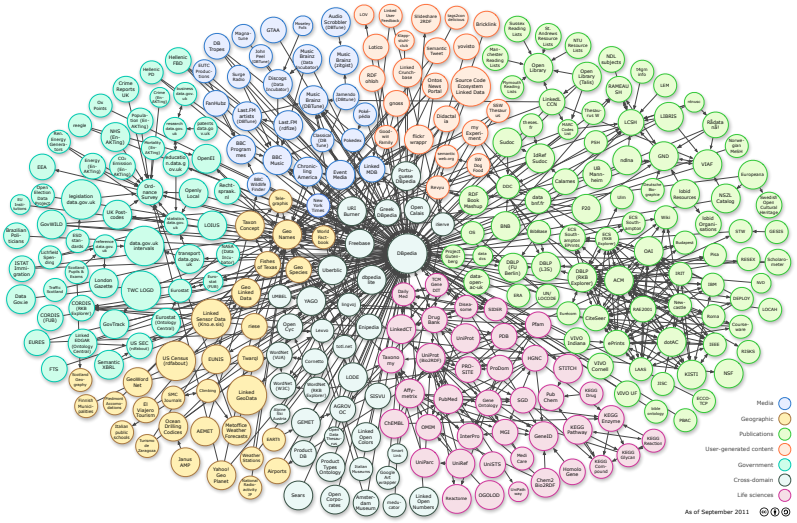


Fig. 1.1 Linking Open Data cloud diagram (taken from [37]).

1.1.1 RDF Data

The Resource Description Framework (RDF) [117], a schema-less, graph-based data format, is used to described the Linked Data model in the form of subject-predicate-object (SPO) expressions based on the statement of resources and their relationships. These expressions are known as RDF triples consisting of three terms that appear multiple times and in any position, in which the *subject* indicates a resource, the *predicate* represents a property of the entity and the *object* is a value of the property in form of a resource or literal. This triple format is very flexible to describe entities in ways that allows it to establish connections between different resources (or literals).

An example of eight RDF triples from DBpedia is shown as Figure 1.2. There, the first three statements convey the information that the Google is a company founded in California and current has 53861 employees, while the fourth one states that California is located in the country United States. Similarly, the remaining four statements present information about IBM and New York.

As stated, the current Semantic Web contains tens of billions of such statements and this number is still rapidly increasing. Actually, even more new facts (statements) could

Triples	
(1)	<dbpedia:Google> <rdf:type> <dbpedia-owl:Company>
(2)	<dbpedia:Google> <dbpedia-owl:foundationPlace> <dbpedia:California>
(3)	<dbpedia:Google> <dbpedia-owl:numberOfEmployees> <53861>
(4)	<dbpedia:California> <dbpedia-owl:country> <dbpedia:United_States>
(5)	<dbpedia:IBM> <rdf:type> <dbpedia-owl:Company>
(6)	<dbpedia:IBM> <dbpedia-owl:foundationPlace> <dbpedia:New_York>
(7)	<dbpedia:IBM> <dbpedia-owl:numberOfEmployees> <434246>
(8)	<dbpedia:New_York> <dbpedia-owl:country> <dbpedia:United_States>

Fig. 1.2 An example of RDF triples.

be inferred when applying the web ontology language such as OWL [87] to existing statements. For instance, from the second and the fourth statement as stated, we can easily infer that Google is founded in the US, which can be represented as a new statement, although this information is implicit. As such kinds of inference has been widely studied in various domains such as *knowledge reasoning* and *artificial intelligence*, this thesis focuses on processing the already large number of explicit statements.

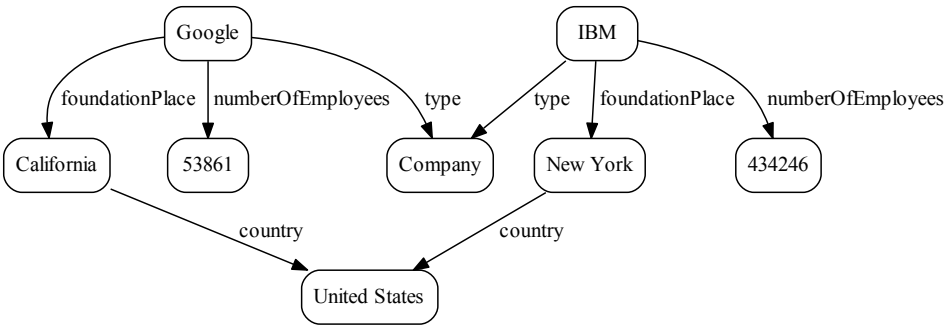


Fig. 1.3 An example of RDF graph.

RDF uses a graph-based data model, a directed graph named as the *RDF graph* [102] can be formulated based on a set of triples. For example, Figure 1.3 demonstrates the graph with the eight triples described in Figure 1.2[1]. In such an RDF graph, all the subjects and objects of each triple is represented as vertices, and the predicate is described as a labeled directed edge from the responsible subject to the object. Note that, all the vertexes in a graph should be kept unique regardless of the number of appearances for a subject or object in the underlying triples. Namely, the same subject or object from different RDF triples is represented by the same vertex.

[1]For simplification, terms of a statement are expressed in an abbreviation form in figures or tables through this thesis.

1.1.2 SPARQL

SPARQL (Simple Protocol and RDF Query Language) is the standard RDF query language that facilitates the extraction of information from stored RDF data. The detailed syntax and semantics of this query language for RDF has been defined by the W3C [118], and the core component of SPARQL queries is a conjunctive set of triple patterns. Similar to an RDF triple, a triple pattern is also in the form of subject-predicate-object, the difference is that any component of the pattern could be a variable. A triple pattern could match a subset of the underlying RDF data, where the terms in the triple pattern respond to the ones in the RDF data [52]. Consequently, a solution mapping is defined as the mapping from the variables to the responsible RDF terms.

```
select   ?x  ?z
where {  ?x <rdf:type> <dbpedia-owl:Company> .
         ?x <dbpedia-owl:foundationPlace> ?y .                    Query 1
         ?y <dbpedia-owl:country> <dbpedia:United_States> .
         ?x <dbpedia-owl:numberOfEmployees> ?z }
```

A simple SPARQL query is shown above as the `Query 1`. This query contains four triple patterns and is used to find out the companies as well as their responsible number of employees, with the conditions that each of the companies should be founded in a place located in the US. If the solution mapping of a triple pattern is defined as μ, on the basis of the eight triples described in Figure 1.2, the corresponding solution μ_i for the i-th triple pattern of the query would be

$\mu_1 :=$ { ?x="<dbpedia:Google>", ?x="<dbpedia:IBM>" }
$\mu_2 :=$ { (?x="<dbpedia:Google>", ?y="<dbpedia:California>"),
(?x="<dbpedia:IBM>", ?y="<dbpedia:New_York>") }
$\mu_3 :=$ { ?y="<dbpedia:California>", ?y="<dbpedia:New_York>" }
$\mu_4 :=$ { (?x="<dbpedia:Google>", ?z="<53861>"),
(?x="<dbpedia:IBM>", ?z="<434246>") }

The solution of a SPARQL query can be formulated based on a series of relational algebraic operators over the solution mappings of each triple pattern according to the syntax of the query [93]. For instance, the variables `?x` and `?y` in `Query 1` appear in different triple patterns imply that there exists *joins* in the process of formulating the final results. On the basis of this, the final result for `Query 1` would be {?x="<dbpedia:Google>", ?z="<53861>"} and {?x="<dbpedia:IBM>", ?z="<434246>"}. Though query operations such as *join*, *sort* and *aggregate* etc. are fully supported by SPARQL, as the *join* is

the most commonly used and also critical for the query performance, we will study this operation and propose novel parallel join algorithms in this thesis.

In addition, similar to RDF graphs, a SPARQL query can also be thought of as a graph called a *query graph pattern*. For example, `Query 1` can be expressed as the graph shown in Figure 1.4. Consequently, a triple pattern is named as a *basic graph pattern*, which describes a subgraph to match against the RDF data. Therefore, the implementation of a SPARQL query is essentially a subgraph matching process. Part of this characterization will be used for data indexing design in this thesis.

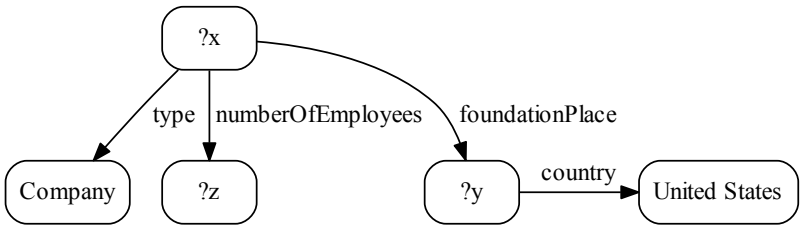


Fig. 1.4 An example of query graph pattern.

1.1.3 RDF Stores

RDF stores are the backbone of the Semantic Web, allowing storage and retrieval of semi-structured information. The engineering of RDF stores is an active area, and various systems and solutions with targets for efficiently processing RDF data have been proposed. As a SPARQL query can be modeled as both rational operations and a query graph pattern, current RDF stores can be consequently represented by two types described as below, depending on their querying processing methods [76].

- *Relation-based RDF stores,* using the logical relational model to store RDF data and translate SPARQL queries into equivalent relational algebraic expressions to execute [23]. In this scenario, the RDF data is normally stored in a set of tables.
- *Graph-based RDF stores,* processing SPARQL queries using subgraph matching algorithms. In this case, the underlying RDF data would be stored as an RDF graph as expressed.

In this thesis, we focus on the relation-based approach and our proposed framework will be based on that as well. The main reasons are: (1) The majority of RDF data management systems is based on the relational method, since it comes with all the benefits of the

mature relational database management systems. In comparison, the graph-based approach is seldom adopted, except for some work on its variants [18, 38, 113], which heavily rely on techniques using graph indexes to reduce the search space of subgraph matching. This indicates that new proposed techniques over the relation-based schema will bring more contributions to the research community. (2) The implementation of relation-based RDF stores mainly uses join operations, whereas graph-based RDF stores use graph exploration for the graph pattern matching. Using join operations, substructures can be joined in batch, which leads relation-based RDF stores to be more suitable for handling large-scale RDF data [110]. This is consistent with our targets to process the big RDF data.

Moreover, RDF systems using the relation-based implementation have repeatedly shown that they are very efficient and scalable in processing RDF data [99]. According to their data structures used for storing RDF data, current solutions can be mainly divided into three categories as follows:

1. *Property table stores,* where a set of property tables is created for stored RDF data. Each table contains multiple RDF properties as attributes, which is modeled as a table column, along with subjects as the table keys.

2. *Triple stores,* where each RDF triple is stored directly in a three-column table, following the form of subject-predicate-object.

3. Other stores, where the underlying RDF data is kept in other formats, with the specified targets for efficient data storage and query execution.

Early RDF stores use the conventional relational database systems (RDBMS) as their underlying stores so as to take advantages of the previous database research on efficient storage and querying [35, 55]. Figure 1.5 shows such an example to store the triples described previously. There, two tables are created, and each of them contains two and one attribute respectively to describe the subjects in the first column. In this case, a SPARQL query would be converted to SQL in the higher level RDF layers, and then sent to the RDBMS which will optimize and execute the SQL query during the query execution [3]. Because relational database management systems are not specifically optimized for processing the semi-constructed RDF data, they encounter bottlenecks both on storage and query for large-scale RDF data - the detailed issues have been presented in [3]. Regardless, several research groups are still working on novel mechanisms to shred RDF into relational and novel query translation techniques to maximize the advantages of this shredded representation as as to improve the query performance [17].

Company

Name	FoundationPlace	# Employees
IBM	New York	434246
Google	California	53861

Place

State	Country
California	United States
New York	United States

Fig. 1.5 RDF data stored as property tables.

Subject	Predicate	Object
California	country	United States
Google	foundationPlace	California
Google	numberOfEmployees	53861
Google	type	Company
IBM	foundationPlace	New York
IBM	type	Company
IBM	numberOfEmployees	434246
New York	country	United States

Fig. 1.6 RDF data stored as triples in a big table.

Compared to the above, triple stores are much more popular and various mature systems have been developed. An intuitive way to store RDF data is demonstrated in Figure 1.6, where each RDF triple is stored directly in a three-column table according to its three terms, and triples are normally sorted according to the value of their subjects. This storage scheme has been widely studied for RDF processing [1, 3, 103, 108]. The reason is that this method shows a flexible way to represent the RDF data: (1) triples can be easily inserted in the table without changing any data structures and (2) solution mappings for each triple pattern can be retrieved by looking up the table. However, there is a potential performance issue for query executions for such stores. The reason is that there is only one single RDF table and there would be many self-joins during query executions, which could be very expensive and thus impacts the query performance. To avoid this problem, popular RDF engines like Jena [86], Sesame [19], RDF-3X [89] and Virtuoso [44] are optimized for SPARQL processing. They create a set of indexes (in the form of SPO, POS and OPS etc.) so as to remove the expensive self-joins as well as to support various query patterns. Additionally, the most popular commercial RDBMSs such as Oracle and DB2 have also supported RDF processing using a similar way [9, 82].

Apart from the two kinds of stores described above, researchers have proposed several novel data structures to store RDF data as well. Among the solutions, stores based on the *vertical table* are shown to be an efficient way for processing RDF data and have been widely discussed [3, 4]. In a vertical table store, the RDF triples are partitioned vertically according to their predicate values and matching triples can be retrieved for triple patterns

type

IBM	Company
Google	Company

numberOfEmployees

IBM	434246
Google	53861

foundationPlace

IBM	New York
Google	California

country

New York	United States
California	United States

Fig. 1.7 RDF data stored as vertically tables.

with predicate constants. In more details, the triples are decomposed and placed into n two-column tables (n is the number of unique properties). In each of these tables, the first column contains the subjects that define that property and the second column contains the object values for those subjects data. In the meantime, each table is sorted by subject, so that particular subjects can be located quickly, and that fast merge joins can be used to reconstruct information about multiple properties for subsets of subjects. For instance, Figure 1.7 demonstrates the vertical tables used for storing the eight triples described in Figure 1.2. Data storage used in our framework will be based on such a scheme, and the details will be given in Chapter 8.

In fact, terms of a triple are always long strings (rather than those shown as simplified examples in Figure 1.2) and many RDF stores normally do not store entire strings in their data tables because of the space consumption and computation overhead. Instead, they store the RDF data on the file system directly in their own binary representation. For instance, Jena [86] and Sesame [19] map strings to integers (ids) so the data is normalized into two tables, one *triple table* keep the content of triples in the form of ids for high-level operations such as querying or reasoning, and one mapping table store the maps of ids and their corresponding strings for string-id and id-string conversion. We will apply this conversion process in our system. More precisely we use the method of *parallel dictionary encoding*, which will be presented in Chapter 5.

1.2 Objectives of this Thesis

As the quantity of available data in the Semantic Web is huge and still increasing at a rapid pace, the corpora of this web has been taken from a lab setting to a terabyte-scale, leading to RDF data becoming deep (complex processing) and reactive (rapidly changing information). Therefore, similarly to other Big Data problems, analytics over such big RDF data brings us

to a new level of computational complexity and consequently becomes difficult to process using traditional approaches.

Many standalone RDF data management systems have been introduced, however, as the size of RDF data increases, such single-machine approaches meet performance bottlenecks, in terms of both data loading and querying. Such bottlenecks are mainly due to (1) limited parallelism on symmetric multi-threaded systems, (2) limited system I/O, and (3) large volumes of intermediate query results producing memory pressure. Therefore, a massively parallel framework over tens, hundreds or even thousands servers becomes desirable. Although several approaches for distributed RDF data processing have been proposed, along with clustered versions of more traditional approaches, as described in our related work in Chapter 2, their techniques operate on a trade-off between loading complexity and query efficiency in the presence of big RDF data.

The objectives (or tasks) of this thesis can be divided into two main parts: (1) Study detailed implementations of current triple stores through system-level characterizations and consequently propose our parallel analytical framework[2] for RDF data processing. Meanwhile, hash tables are the most commonly used structure in data processing, and we investigate efficient parallel hash algorithms in the presence of large-scale data so as to support high-performance implementations of our system. (2) From the basis of (1), we propose new parallel approaches/techniques for detailed implementations of each phase of the proposed framework, improve their performance and consequently achieve fast loading and querying of large-scale RDF data on distributed infrastructures.

For the core part of this thesis, namely the second objective, we will focus on proposing approaches with *full parallelism* and *distributing everything* rather than high-level operations such as task scheduling or thread coordination etc. on a distributed system. The reason is that we are more interested in **exploring** and **applying** new efficient parallel techniques for managing huge RDF data. In such scenarios, to achieve a high performance RDF data analytical system, we have to address the following three core challenges:

- Computation: a very large number of data intensive operations such as lookup and joins could potentially be generated, efficient strategies are required to simplify or reduce such operations so as to reduce core utilization and minimize energy consumption during system implementations.

- Communication: a very large number of points of the distributed dataset would be potentially accessed, efficient algorithms which exploit locality of access are required in

[2]We also refer to it as a system as we have conducted a general implementation in Chapter 8.

order to minimize data movement and message traffic during system implementations.

- Load Balancing: real-world Linked Data is highly skewed [78] while operations over such data would lead to load imbalancing, efficient approaches are required to remove computation hotspots so as to improve the horizontal scalability of the system.

1.3 Contributions of this Thesis

This thesis aims to develop a distributed analytic framework for fast processing large RDF data, in terms of data loading and querying. During this process, a number of original contributions were produced as following.

Pre-studies and Analysis

Before the design of our framework, we first studied the detailed implementations of current triple stores through systematical-level experimental evaluations. Then, we also designed and evaluated parallel hash algorithms for large-scale data over a distributed system. For this part, the main contributions are:

1. To allow the dynamics and behaviors of query execution for RDF stores to be better understood and so help in the design of efficient distributed systems, optimized for parallel RDF processing, a detailed experimental analysis of four of the most popular and mature triple stores has been conducted. We construct suitable system-level metrics and implement our experiments on different platforms. To the best of our knowledge, this is the first time in the literature that anyone has reported on the performance and characteristics of triple stores on an enterprise platform. This work was published in [25, 26].

2. Since high-performance analytical data processing systems often run on servers with large amounts of memory and hash tables are the most common used data structure in such environments, a high-level structured framework of parallel hashing designed for processing massive data is proposed. Different to conventional approaches, this framework supports both distributed memory while avoiding frequent remote memory access, and thread coordination on a per-partition basis. From there, an efficient parallel hashing algorithm which employs the popular *compare-and-swap* (CAS) and the proposed *range-based* lock-free hashing strategies is presented. The experimental evaluation results show that our implementation is highly efficient and scalable for

processing large datasets. Also, the proposed *range* strategy for our hashing implementation is faster than the popular used CAS operations within the proposed framework. This work was published in [28].

Design and Evaluations

Based on the studies in triple store and parallel hashing, a parallel framework for analyzing large RDF data is proposed as in Figure 1.8. The whole data process is divided into two parts - data loading and data querying. (1) The raw RDF data at each computation node (core) is encoded in parallel in the form of integers and then loaded in memory in local indexes (without redistributing data). (2) Based on the query execution plan, the candidate results are retrieved from the built indexes, and parallel joins are applied to formulate the final outputs. In the latter process, local filters at each node can be used to reduce/remove the retrieved results that have no contribution for the final outputs, and the redistributed data during *parallel joins* can be used to create additional sharded indexes.

Different from a centralized or a sequential distributed structure, here we highlight that the data processing in each step in our framework is fully parallel. To catch the core performance issues of an RDF system, this thesis concentrates on the parallel techniques used for data encoding, parallel joins and data indexing. The detailed contributions here are:

3. To avoid unnecessarily high disk-space consumption and reduce complex computation during query executions, a scalable solution for dictionary encoding massive RDF data in parallel is proposed. A detailed implementation with several optimizations using the *asynchronous partitioned global address space* model programming language - X10 [22] is presented. Moreover, a performance evaluation with up to 384 cores and with datasets comprising of up to 11 billion triples (1.9 TB) is conducted. Compared to the state-of-the-art approach [116], the proposed approach is faster (by a factor of 2.6 to 7.4), can deal with incremental updates in an efficient manner (outperforming the state-of-the-art by several orders of magnitude) and also supports both disk and in-memory processing. This work was published in [34].

4. To efficiently handle data skew and thus reduce the load-imbalancing during parallel join operations, a novel approach, *query-based distributed join*, is proposed for processing large-large table skew joins on distributed architectures. We present the detailed implementation of our method and conduct an experimental evaluation over

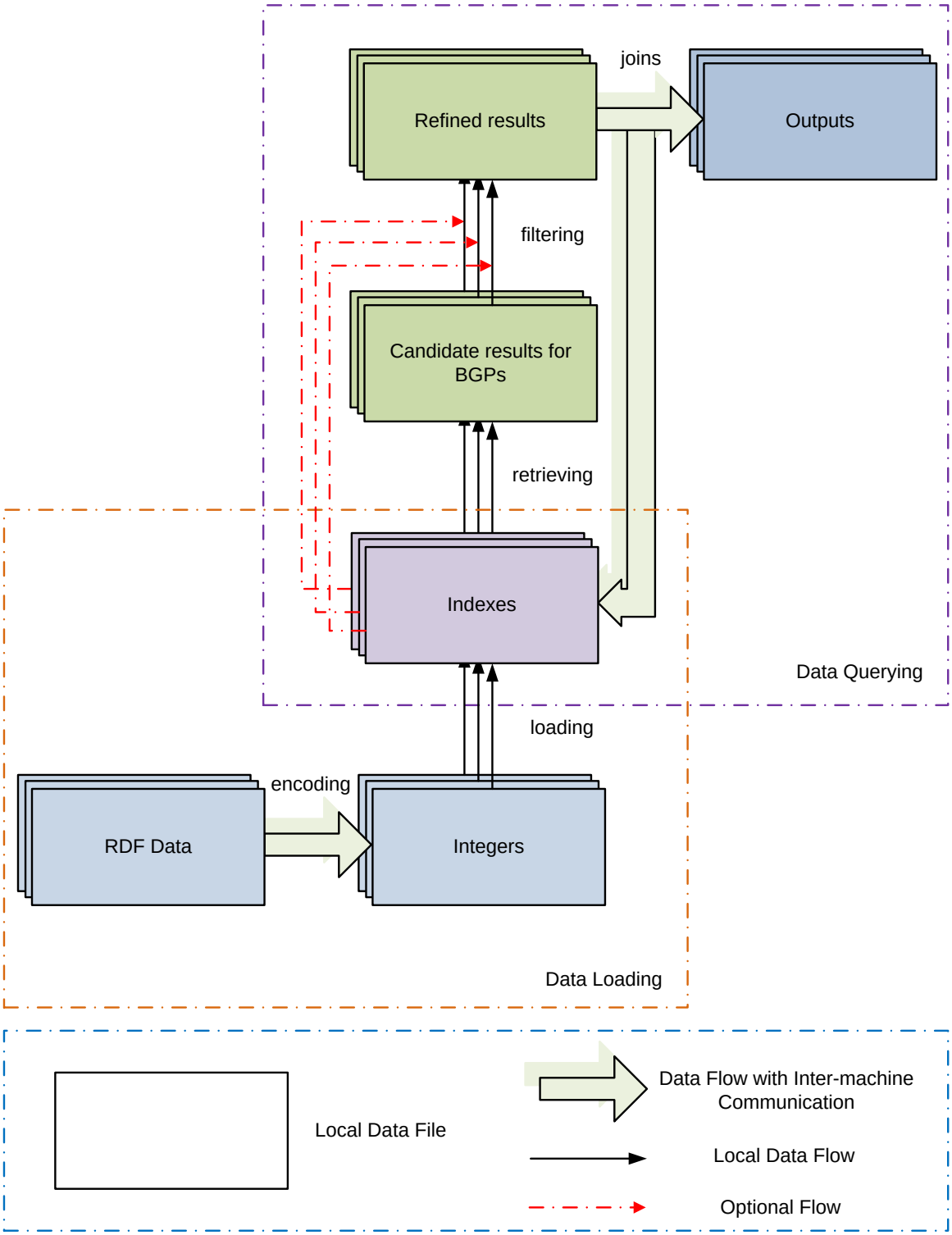


Fig. 1.8 General design of our parallel framework, which includes two main parts, the data loading and data querying. This thesis focuses on the techniques used in three core parts for a system: encoding, joins and indexing.

a commodity cluster of 192 cores (16 nodes) and datasets of 1 billion tuples with different skews. The results show that the proposed method is scalable, and also runs faster with less network communication compared to the state-of-art approach [127] under high data skews. An extended outer join algorithm on that basis is also introduced and is also shown to be able to outperform the state-of-art techniques [126, 127] under high skews, which includes special optimization for outer joins. This work was published in [27, 29].

5. To further improve the robustness and efficiency of the *query-based distributed joins*, a new parallel join algorithm referred to as PRPQ (*partial redistribution & partial query*) is introduced. We conduct a detailed theoretical performance comparison between this method and the state-of-art method [127]. We also present the detailed implementation and evaluation of the proposed method. The experimental results demonstrate that the proposed PRPQ algorithm is indeed robust and scalable under a wide range of skew conditions. Specifically, compared to [127], our algorithm is always faster, and a notable performance improvement with less network communication has been achieved under different workloads, figures that confirm the theoretical analysis. This work was published in [31, 32].

6. To achieve fast loading and querying over large-scale RDF data, a distributed RDF data indexing method is proposed. Based on a simple similar-size data partitioning infrastructure, a dynamic two-tier index approach is presented and the design of a pair of performance-enhancing distributed filters is also introduced. Experimental results on a commodity cluster of 16 nodes show that our multi-level indexing approach can indeed highly improve loading speeds while remaining competitive in terms of performance. Our system can load a dataset of 1.1 billion triples at a rate of 2.48 million triples per second and provides competitive query performance to current RDF systems RDF-3X [89] and 4store [54]. This work was published in [30, 33].

Additional Contributions

In fact, the proposed techniques for RDF data encoding, joins and indexing in our framework can be also independently applied to other data problems. For example, the proposed join algorithms can be used for computing the *well-founded semantics* over big data. In this case, we have conducted an experimental evaluation for various rule sets and data sizes using a basic parallel join method, and the results have shown that the implementation is

highly efficient and can compute billions of facts in minutes using 192 cores. This work was carried out in collaboration with Ilias Tachmazidis, Spyros Kotoulas, Grigoris Antoniou and Tomas Ward. It was published in [111].

1.4 Outline of this Thesis

There are eight subsequent chapters in this thesis, which are organized as follows:

- **Chapter 2** presents a comprehensive review of current RDF data systems (both standalone and parallel solutions) and the related parallel techniques, in terms of dictionary encoding, parallel joins and data indexing. Discussions of such systems (or techniques) are presented as well.

- **Chapter 3** proposes several systematical metrics to characterize the runtime of current triple store implementations. The four most popular systems are evaluated over two different platforms with large numbers of triples. Through detailed time cost and system consumption measures of queries derived from a benchmark, the dynamics and behaviors of query execution of the systems are described.

- **Chapter 4** focuses on investigating efficient parallel hash algorithms for processing large-scale data. A high-level parallel hashing framework, Structured Parallel Hashing, targeting efficiently processing massive data on distributed memory, is proposed and theoretically analyzed. Moreover, two kinds of lock-free strategies within the framework are presented and experimentally evaluated.

- **Chapter 5** describes a very efficient parallel dictionary encoding algorithm for RDF data. The detailed implementation as well as a very extensive quantitative evaluation of the proposed algorithm is presented. At the same time, performance comparison with the state-of-art MapReduce-based method [116] is also provided.

- **Chapter 6** introduces the *query-based joins*, a novel parallel join approach for handling data skew in distributed architectures. From this basis, another new algorithm specified for outer joins referred to as QC (*query with counters*) is proposed as well. The detailed design of both approaches and their performance evaluations are also presented respectively.

- **Chapter 7** proposes a new efficient and robust join algorithm named PRPQ (*partial redistribution & partial query*) based on the idea of Chapter 6. A detailed theoretical

performance analysis with comparison with the state-of-art PRPD algorithm [127] is given. In the meantime, detailed implementation and quantitative evaluation with various join workloads of the proposed approach are also presented.

- **Chapter 8** introduces a two-tier index approach for RDF data on distributed systems, which includes a lightweight primary index and a series of dynamic, multi-level secondary indexes. Further, two kinds of distributed filters to replace the secondary indexes are also proposed so as to reduce memory consumption. On that basis, experimental evaluation of the proposed method as well as performance comparison (both data loading and querying) with current systems on a commodity shared-nothing cluster are presented.

- **Chapter 9** concludes this thesis and highlights future research arising from this work.

Chapter 2

Related Work

2.1 Introduction

We are aiming to apply parallel techniques to RDF data management systems so as to build a scalable RDF analytic framework. With the objectives and also the challenges in terms of system implementation as described in Chapter 1, this chapter reports on related work in the field which can be organized as two main parts as follows.

The first part presents current high performance RDF data management systems and the benchmarks for such RDF stores. As some existing RDF systems will be used as references for the design and evaluation of our framework, we first introduce and discuss current standalone and parallel solutions in Section 2.2. Initally we focus on the novel data structures for indexing and optimized techniques of the former before we examine the distributed indexing approaches for the latter (because index is the pivot for data loading and querying). Then, in Section 2.3, various RDF datasets, popular benchmarks and related evaluation works of RDF stores are presented, because part of them will be used to evaluate the performance of our own implementations.

The second part focuses on the detailed approaches used for RDF data compression and parallel joins. The methods of RDF compression are presented at first in Section 2.4 with an emphasis on parallel dictionary encoding algorithms. Following that, in Section 2.5, detailed parallel inner- and outer join approaches are introduced. More specially, for both the *compression* and *joins*, we describe the detailed implementation of the state-of-art techniques and analyze their possible performance issues. We also conduct a general comparison between these approaches and our proposed methods to be presented in later chapters. Because we use the X10 parallel programming language [22] throughout this work, a detailed introduction to this language as well as its advantages derived from our experiences

are given in Section 2.6.

2.2 High Performance RDF Data Management Systems

2.2.1 Sequential Solutions

Significant efforts have been dedicated to the development of solutions for RDF data management. Along this line of research, some methods have achieved high performance on processing RDF data over a single computational node, either by designing novel structures for the underlying data or applying database optimization techniques to data storage and querying. Here, we introduce some typical approaches.

Novel Data Structures

SW-Store. SW-Store [4] stores the RDF data in two-column tables, using the vertical partitioning method as described in Chapter 1, so that the candidate results of a triple pattern can be fast located by looking up the properties. In the meantime, as the first column (subject) of each table is sorted, high performance table-merge operations can be facilitated during query execution. The experimental evaluations show that SW-Store performs very fast on RDF querying over a column-oriented DBMS such as the C-store [108], implementing queries in seconds over 50 million triples. In comparison, a common property table store or a triple store takes hundreds of seconds [4].

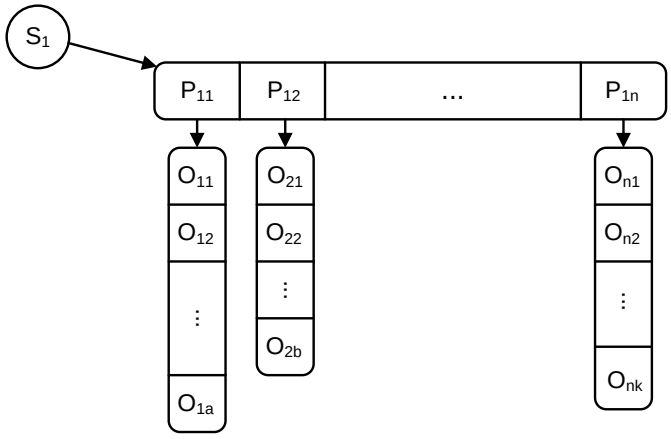


Fig. 2.1 An example of SPO indexing in a Hexastore [121].

Hexastore. As the vertical partitioning method can suffer from scalability drawbacks for queries that are not bounded by RDF properties, Hexastore [121] provides another structure

to enhances the vertical partitioning idea and takes it to its logical conclusion. In this store, RDF data is indexed in six possible ways, to account for all possible orders of precedence of the three RDF terms. In addition, the data is kept in a set of vectors based on the nature of triples and each term of a triple is associated with two vectors, one for each of the other two terms. Moreover, lists of the third RDF element are appended to the elements in these vectors. An example for the SPO index[1] is shown as Figure 2.1, where the first list stores the triples <S_1 P_{11} O_{11}> and <S_1 P_{11} O_{12}> etc. Similar to the vertical partitioning method, this data format allows us to quickly locate the solution mappings for each triple pattern. Moreover, the multiple indexing structure has also significant advantages compared to previous approaches on processing different subgraph patterns. The experiments show that Hexstore is scalable for processing general-purpose queries, and it achieves orders of magnitude better performance than the column-oriented vertical-partitioning methods, although it comes with the price of a worst-case five-fold increase in index space.

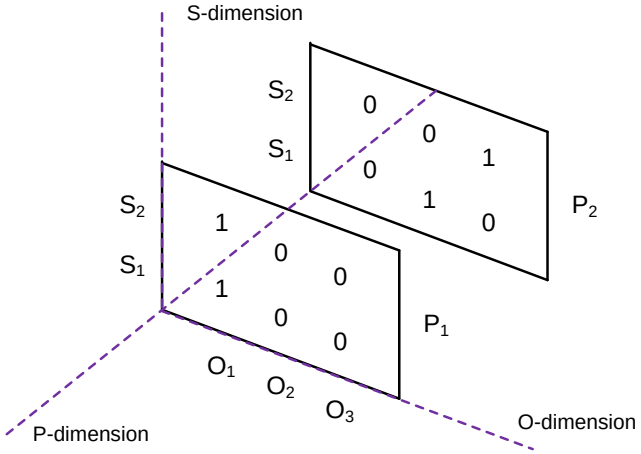


Fig. 2.2 An example of bit-matrix structure for storing RDF data in BitMat [11].

BitMat. A bitmap index is an efficient way to store the bulk of its data in the form of bit arrays and answer queries by performing bitwise logical operations over these bitmaps [21]. Applying this idea to RDF data management, the BitMat method [11] represents RDF statements by a 3D bit-cube, where each dimension indicates the subjects (S), predicates (P) and objects (O). An example is shown in Figure 2.2, according to the dimension of predicates, the triples in the upfront layer are <S_1 P_1 O_1> and <S_2 P_1 O_1> while the triples in the latter lay are <S_1 P_2 O_2> and <S_2 P_2 O_3>. As the data can be efficiently compacted based on this method, this approach can be used to store large RDF data sets. In the meantime, the basic

[1] Throughout this work, when we refer to index, the S, P and O is responsible for the subject, predicate and object of RDF data.

join can be simply implemented by logical bitwise `and/or` operations on parts of a BitMat. Furthermore, compared to a general query execution, the memory consumption of this solution can be highly reduced because of the intermediate and final results in a multi-join do not need to be completely materialized. The experiment results demonstrate that the BitMat store can process up to 1.33 billion triples – the best result published for a single-node solution. It has also achieved completed query performance with RDF-3X [89] with highly selective queries and can deliver 2-3 orders of magnitude better performance on complex, low-selectivity queries over massive data.

Database Optimization

RDF-3X. RDF-3X [89] is an open source RDF engine. The same as other triple stores, it converts the terms of each triple from Uniform Resource Identifiers (URIs) or literals to integer IDs using a mapping dictionary. Namely, the RDF statements are stored as ID triples. To achieve fast processing ability on RDF data, RDF-3X provides three main optimization as follows, on data storage and query execution:

1. Indexes over all 6 permutations (SPO, POS and OPS etc.) as well as 9 aggregated indices (SP, PO and P etc.) are built so that the results for all the triple patterns in any ordering can be quickly looked up using range scans. Additionally, all the indexes are efficiently compressed by a delta-based byte-level compression scheme. This scheme exploits the fact that it usually takes fewer bytes to encode the delta between triples than to store the triples directly, which makes that total storage space consumed by all the indexes less than the size of the original data.

2. Two types of join operators, hash join and merge join, are employed by the engine, and the query processor leverages fast merge joins to the largest possible extent. Namely, if both inputs of a join operator are ordered by columns corresponding to the join variable, then merge join will be used; otherwise, the hash join is used [76]. This highly improves the performance of joins appearing frequently in queries and makes the query execution of RDF-3X much faster than other triple stores, such as Jena, which uses the nested-loop joins.

3. A query optimizer is used to formulate the optimal join orders for the query execution plans so as to reduce the intermediate results and consequently reduce the join costs. It employs dynamic programming for plan enumeration, with a cost model

based on RDF-specific statistical synopses. These statistics include counters of frequent predicate-sequences in paths of the data graph; such paths are potential join patterns. Compared to the query optimizer in a universal database system, the proposed optimizer is simpler but much more accurate in its selectivity estimations and decisions about execution plans [89].

Additionally, in extended work [91], the authors of RDF-3X integrate another technique, *sideways information passing* (SIP) [68, 105], into their system. This approach can efficiently reduce the inputs of a join operator outside the normal execution flow. Namely, following a query plan, the redundant intermediate results, which have no contribution for the final outputs, will be removed during query execution, and thus the query performance is further improved. All these designs make RDF-3X known as the fastest RDF engine, and we will conduct a performance comparison with this system in Chapter 8.

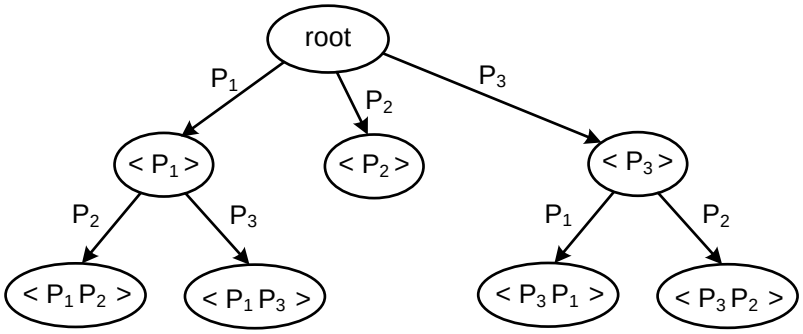


Fig. 2.3 An example of the tree of predicate path that is used for filtering non-useful results during query executions.

R3F. Most recently, the work [76] discusses the statistical and the SIP techniques used in RDF-3X and argues that the two approaches used for handling the intermediate results have a limitation that they do not consider any graph structures in RDF data. Then, it introduces a new method, *RDF triple filtering* (R3F), to exploit the graph-structural information of RDF data so as to be more efficient in reducing the vast number of redundant intermediate results during query processing.

The core component of R3F is a path-based index called *RDF path index*, for efficiently filtering triples, which have no contribution for the final outputs, before join operations. An example of such an index represented in the form of a tree is shown as Figure 2.3. Each node of the tree contains the associated vertexes following the specified predicate path of the RDF graph. In fact, the building process of such an index tree is very similar to a kind of pre-computing of solution mappings for a set of SPARQL queries. For instance, the

leftmost node in the given tree only contains the unique terms which match the path <P_1 → P_2>, and these terms are actually the unique results of the `Query 2` shown as below. From this basis, when implementing a query containing the subgraph of `Query 2`, then the number of solution mappings {`?y ?z`} of the second triple pattern can be efficiently reduced by checking the existing unique values of `?z` before they take part in a join. This means that the node in the index tree can be consider as a filter for queries following the specified predicate path. Moreover, when the stored triples are sorted, this filter can also remove the unnecessary range scans during result lookup.

```
select  ?z
where   { ?x P1 ?y .  ?y P2 ?z . }     Query 2
```

Since the size of the path-based index could be huge (because it is based on all possible graph paths), R3F has employed various techniques on reducing the space consumption. Additionally, a relational operator that can conduct the triple filtering with little overhead compared to the original query processing is proposed as well. The authors of R3F have integrated the proposed new techniques into the RDF-3X engine, and their experiments on large-scale RDF datasets demonstrate that the presented methods can efficiently reduce the number of redundant intermediate results and obviously outperform the original RDF-3X implementations during query processing. We will also employ some of these ideas in our framework in Chapter 8 and build efficient distributed RDF filters so as to improve the query performance.

2.2.2 Parallel Solutions

The sequential implementation presented above are shown to be efficient on processing RDF data, regardless, they have not adopted any parallel techniques so as to use the advantages of modern multi-core architectures. Further, with continuous increasing of the amount of published RDF data, as stated previously, the performance of data loading and querying will be limited by the underlying platforms. Therefore, a parallel RDF system, which aims to improve performance through parallelization of various operations such as building indexes and evaluating queries, becomes much more attractive.

Several distributed RDF data processing systems have been proposed. Like the single-node solutions, the index itself in most of these systems also contains all the data. Depending on their data partitioning and placement patterns, we divided current distributed solutions into four main categories as following. We will present them with details in turn and also discuss their advantages and disadvantages in the presence of large-scale RDF data.

To better understand the basic idea of each approach in the following descriptions, we take an simple example, including four triples and two queries, which is shown in Figure 2.4 and Figure 2.5 respectively. We present the detailed implementation of each method over a two-node system and assume that terms with an odd number hash to the first node and constants with an even number hash to the second node (e.g. `B1` hashes to node 1, `B2` hashes to node 2) through out such examples in this thesis.

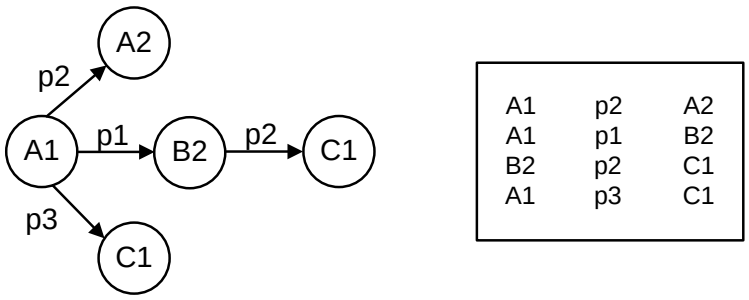


Fig. 2.4 An RDF graph and the responsible triples.

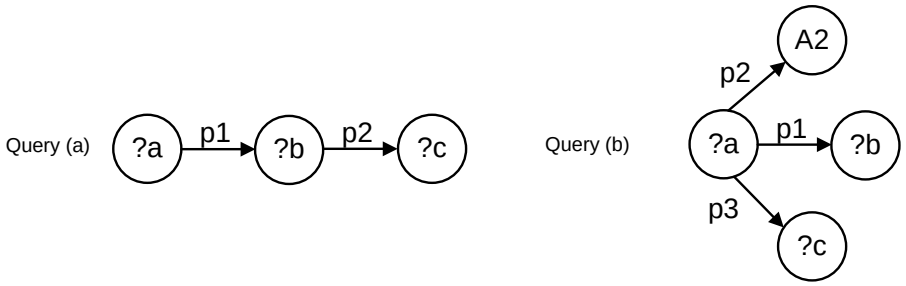


Fig. 2.5 Two queries in the form of graph patterns.

Similar-size Partitioning

Systems based on similar-size partitioning place similar volumes of raw triples on each computation node without a global index. During query processing, nodes provide bindings for each triple pattern can be implemented in parallel, and the intermediate (or final) results can be then formulated by *parallel joins*. Figure 2.6 shows the details of the partitioning that each node will hold two triples. Then during query execution, the solution mapping of each triple pattern will be located to a same node to implement local joins and consequently formulate the intermediate or final results. For example, for the `Query(a)` in Figure 2.5, the result of the first triple pattern `<?a p1 ?b>` at the first node `<A1 B2>` will be transferred to the second node, based on the hash value of the join key `B2`, to join with the `<B2 C1>` at the

second node, and then output of the query <A1 B2 C1>. In current literature, the work [120] is fully based on this approache and [79] employs some additional skew-handling techniques for join operations during query execution.

Fig. 2.6 The similar-size partitioning method over a two-node system.

It can be seen that this schema has obvious performance advantages on data loading, as similar-size is very easy to achieve and each computing node can simply load its local data in parallel without inter-node communications. Regardless, for any query including join operations, there will always be data movements in the specific implementations, which can consequently decrease the query performance.

Hash-based Partitioning

Exploiting the fact that SPARQL queries often contain *star* graph patterns, triples under this scheme are commonly hash partitioned (by subject) across multiple machines and accessed in parallel at query time. As shown in Figure 2.7, the three triples with subject A1 are assigned to the first node while the other is assigned to the second node. Clearly, this kind of assignment will be more time cost than the above *similar-size* method, and there also exist same data movements when implementing the Query(a). However, when a query containing *star* pattern, the Query(b) in the figure for instance, then the included join operations will be totally computed locally, which can efficiently reduce the costly network communications and consequently improve the query performance.

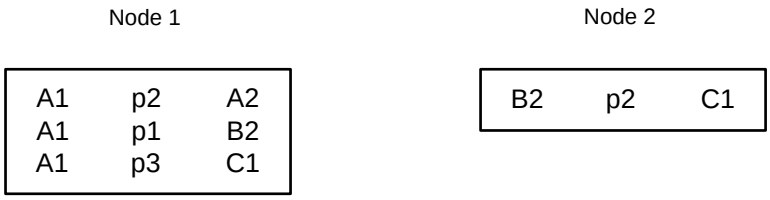


Fig. 2.7 The hash-based partitioning method over a two-node system.

SHARD [97] and the solution [63] are on the basis of such a schema, both are implemented using the MapReduce model so as to process large RDF graph. In SHARD, the RDF data is persisted in the flat files in the HDFS that each line of the triple-store text file

represents all triples associated with a different subject. During the query processing, an iteration of map-reduce-join continues until all clauses in a query are processed and variable are assigned. Compared to this, to achieve higher query performance, [63] employs a more refined partition method. The N-triple data is firstly divided according to their predicates. Then, the *rdfs_type* file is divided into as many files as the number of distinct objects the predicate `rdf:type` has, and other predicate files are divided according to the type of the responsible objects. In addition to this, an algorithm about how to schedule the jobs of a query is also provided.

Sharded/Partitioned Indexes

This approach is very closed to the centralized stores, triple indexes in the form of SPO, OPS etc. are distributed across all the computing nodes and stored locally as a B-Tree. Most of the existing parallel systems such as YARS2 [56], Clustered-TDB [92], Virtuoso-cluster [44], BigData [112] and 4store [54] belong to such a schema. Their operations are more similar to single-node RDF stores, normally offering lower loading speeds but can achieve persistence and more space-efficient indexing over a distributed system. Meanwhile, system I/O and join throughput of queries can be improved as well on that basis.

Graph-based Partitioning

Graph partitioning algorithms are used to partition RDF data in a manner that triples close to each other can be assigned to the same computation node. SPARQL queries generally take the form of graph pattern matching so that sub-graphs on each computation node can be matched independently and in parallel, as much as possible. Using such method, all the previous four triples will be placed on the same node based on a 2-hop graph (namely distance between two node is 2 maximum) as shown as Figure 2.8. Compared to the three approaches above, it can be seen that there will be no network communication for such a method during query executions, for both the queries in Figure 2.5. However, as graph partitioning is always complex, especially for large graph, the connections between each node will increase exponentially with increasing the graph, which could induce a very large time cost before loading the data.

The most typical solution under such a schema is [61]. Using the graph-partitioner METIS [73], the approach [61] partitions the RDF data based on its vertexes, such that each machine in a cluster receives a disjoint subset of RDF vertexes that are close to each other in the graph. During the triple placement, to minimize the network communication

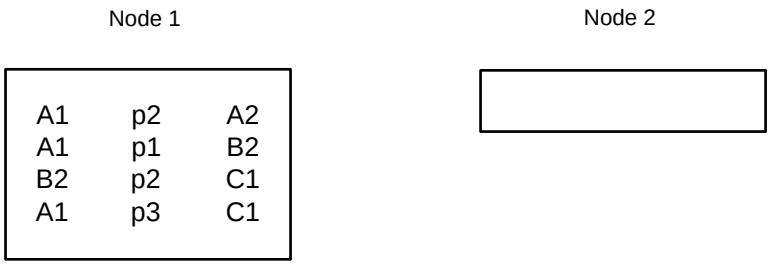


Fig. 2.8 The graph-based partitioning method over a two-node system.

some triples on the boundary is replicated across partitions, and the n-hop guarantee method is used to specify the amount of overlap. Moreover, based on the data partitioning, the SPARQL queries are also decomposed into high performance fragments so as to maximize the parallelism without communication as possible.

In general, the four different approaches outlined above operate on a trade-off between loading complexity and query efficiency, with the earlier ones offering superior loading performance at the cost of more complex/slower querying and the latter ones requiring significant computational effort for loading and/or partitioning. In contrast to this, as we will present in Chapter 8, our proposed dynamical indexing approach can both load and query large RDF data very quickly.

2.3 RDF Store Benchmarks

2.3.1 RDF Benchmarks

As RDF data management systems are proliferating, a number of benchmarks specified for these systems have been proposed so as to test their scalability and performance under data and workloads with various characteristics [42]. Currently, there are four main RDF benchmarks as follows.

The Berlin SPARQL Benchmark (BSBM) [15] features an e-commerce use case in which a set of products is offered by different vendors and consumers have posted reviews about products. Different from other benchmarks, BSBM can generate RDF data in three models (*RDF triple data model*, *named graphs data model*, *relational data model*) with same semantics. BSBM also provides three different query mixes for the purposes of testing different common use cases. Compared with other benchmarks, the BSBM queries are much more complex in terms of RDF workload. We will investigate the inside detailed implementations of current popular triple stores in Chapter 3 from the basis of this benchmark.

The Lehigh University Benchmark (LUBM) [53] is the most widely used benchmark. It adopts a `University` domain and describes universities, departments and the activities that occur at them. The LUBM benchmark provides 14 extensional test queries representing a variety of properties. As the queries are relatively simple, as they do not contains any aggregation syntax, which is suitable for evaluation of the core performance of query execution, we will use it to evaluate our framework in Chapter 8.

The SP2Bench benchmark (SP2Bench) [100] uses the DBLP as a domain for the dataset. Therefore, the types encountered include Person, Inproceedings, Article and the like. The Person type is the most instantiated in the dataset, as is the case for the name and homepage properties. The SP2Bench benchmark is accompanied by 12 queries.

The DBpedia SPARQL benchmark (DBPSB) [88] applies to the DBpedia knowledge base and procedures for benchmark creation is based on query-log mining, clustering and SPARQL feature analysis. A set of 25 SPARQL queries is derived as templates, which cover most commonly used SPARQL features and are used to generate the actual benchmark queries by parametrization.

2.3.2 Benchmark Datasets

Except for the datasets derived from the full RDF benchmarks as described, various independent RDF datasets have been also used for evaluating RDF systems (mainly on RDF compression and reasoning). The most popular data are, BTC (Billion triple challenge), Uniprot [10], YAGO [109] and Barton Library Dataset [2].

BTC is a web crawl encoding statements in the form of N-Quads, which consists about 2.2 billion statements currently. Uniprot [10] is a large collection of biological function of proteins derived from the research literature, containing 6.1 billion triples and still increasing. In comparison to this, the YAGO dataset [109] and the Barton library dataset [2] are much smaller. The former brings together knowledge from both Wikipedia and Wordnet, and currently consists of about 19 million triples. The later one consists of approximately 45 million RDF triples that are generated by converting the Machine Readable Catalog data of the MIT Libraries Barton catalog to RDF. We will use the first two big datasets in our evaluations in Chapter 5.

2.3.3 Evaluation Work

Along with the growth in new RDF store implementations, there has been a corresponding increase in interest for relevant performance evaluations. Liu et al. [81] evaluated 7 RDF

stores by comparing data loading and query response time over different size datasets generated using the LUBM benchmark. Rohloff et al. [96] implemented the queries and datasets from LUBM to compare the performance of triple stores with different storage backends (such as MySQL etc.) by applying metrics like load time, query response time, query completeness and soundness, and disk-space requirements. Schmidt et al. [100] compared the performance of a single triple store and the vertically partitioned scheme for storing RDF data in DBMS using their SP2Bench benchmark. Bizer et al. [15] performed an evaluation over different RDF systems with their BSBM benchmark, through comparing the loading time, overall runtime and average runtime per query. Furthermore, Bröcheler et al. [18] presented an experimental assessment of their DOGMA system by comparing the performance with other RDF database systems in many cases, like query time and index size. Most recently, Morsey et al. [88] compared four popular triple stores through measuring *queries per second* (QpS) and *query mixes per hour* (QMpH) over different size datasets on the basis of their DBPSB benchmark.

All these reports have provided valuable insight on the performance of RDF stores. However, all these evaluation experiments operate on an application level but have not gone into the system-level to discover performance inhibitors and bottlenecks. In contrast, we will present a detailed system-level evaluation of current triple stores in Chapter 3.

2.4 RDF Data Compression

As the terms in a RDF statement consist of long string characters in the form of either URIs or literals, storing and retrieving such information directly on an underlying database namely a triple store will result in (1) unnecessarily high disk-space consumption and (2) poor query performance (querying on strings is computationally intensive).

Compression has been extensively studied in various database systems, and has been considered as an effective way to reduce the data footprint and improve the overall query processing performance [1, 24, 69, 122]. In terms of efficient storage and retrieval of RDF data, the approaches described in [45] are geared toward efficient storage and transfer, as opposed to having direct access to the data for efficient processing. In comparison, the approach *dictionary encoding* has been shown to be an efficient way to ameliorate these problems. In conventional dictionary encoding approaches, for all elements, their ids are retrieved (or generated) through the sequential access of a single dictionary. This method is easy to implement, and is commonly adopted by the current triple stores such as RDF-3X [89] etc. Regardless, it does not avail of potential speed-up by parallel implementations

and is not suitable for compressing large data sets due to time considerations and memory requirements. Consequently, encoding triples in parallel based on a distributed architecture with multiple dictionaries, becomes an attractive choice for this problem.

Though various distributed solutions used to manage RDF data have been proposed in the literature [56, 61], their main focus is on data distribution after all the statements have been encoded. To our knowledge, currently there exists only two efficient methods focused on parallel dictionary encoding of RDF data. One is based on parallel hashing [51] and the other uses the MapReduce model [116].

Goodman et al. [51] adapt the linear probing method on their Cray XMT machine, and realize the parallel encoding on a single dictionary through parallel hashing, exploiting specialized primitives of the Cray XMT. Their evaluation has shown that their method is highly efficient and the run-time is linear with the number of used cores. This method requires that all data is kept in memory and is deeply reliant on the shared memory architecture of the Cray XMT, making it unsuitable for commodity distributed memory systems. They report an improvement by a factor of 2.4 to 3.3 compared to the MapReduce system on an in-memory configuration. By comparison, on similar datasets, our approach outperforms the MapReduce system by a factor of 2.6 to 7.4, both on-disk and in-memory.

Compared with [51], the MapReduce method proposed by Urbani et al. [116] is more general in that it can be run on ordinary clusters and on-disk. There are three main elements to their system: (1) the popular terms are cached in memory by sampling the data set, so that these popular terms assigned to each task could be encoded locally and consequently prevent eventual load balancing problems, (2) a hash function is used to assign grouped terms to reduce tasks, which then assign the term identifier, keeping the consistency of the encoding, and (3) the MapReduce framework facilitates the parallel execution of the program. Although their evaluation on the Hadoop framework has shown that their system is efficient and scales well, as we will show in Chapter 5, our approach is much faster and more flexible, and also support both disk and in-memory implementations.

2.5 Parallel Join Approaches

Data warehouses and the web comprise enormous numbers of data elements and the performance of data-intensive operations on such datasets, for example for query execution, is crucial for overall system performance. *Joins*, which facilitate the combination of records based on a common key, are particularly costly and efficient implementation of such operations can have a significant impact in improving the performance on a wide range of

workloads, ranging from databases to decision support and Big Data analytics.

A significant corpus of research in parallel joins on shared-memory systems has already achieved significant performance speedups through improvements in architecture at the hardware level [7, 20, 75]. Nevertheless, as applications grow in scale, the associated scalability is bounded by the limit on the number of threads available and the availability of specialized hardware platforms. Though GPU computing has become a well-accepted high performance parallel paradigm and there are many reports on GPU implementations of parallel joins [57, 71], as in shared-memory architectures, when the scale of data is high, the memory and I/O eventually become bottlenecks. As a consequence, the efficient parallelisation of join on distributed memory machines becomes increasingly desirable.

In this section, we introduce the conventional approaches for parallel joins, including both inner- and outer joins. We discuss their potential performance issues in the presence of large scale data, especially in the case of skewed data. Moreover, related techniques and state-of-the-art methods, which can efficiently handle the data skew, are also presented and discussed.

2.5.1 Inner Joins

The *join*[2] is one of the most popular operation used in various data management systems. It combines two relations based on a common *join key*. For example, the join between a relation R with attribute a and another relation S with attribute b, is evaluated by the pattern $R \bowtie S$ where $R.a = S.b$.

Basic Approaches

Various distributed join algorithms have been proposed [39, 66, 77, 101, 119, 127], all of which can be considered variations of two fundamental distributed frameworks: hash-based and duplication-based joins. Such approaches can be broadly decomposed into an initial distribution stage followed by a local join process. This latter process is well studied and techniques such as the sort-merge join and the hash join are commonly used. We have selected the hash-join as the local join process for our analysis. To capture the core performance of queries, we focus on exploiting the parallelism within a single join operation between two input relations R and S over an n-node system, assuming both R and S are in the form of <*key, value*> pairs and $|R| < |S|$ in the following.

[2]Unless otherwise stated, a *join* means an inner join throughout this thesis.

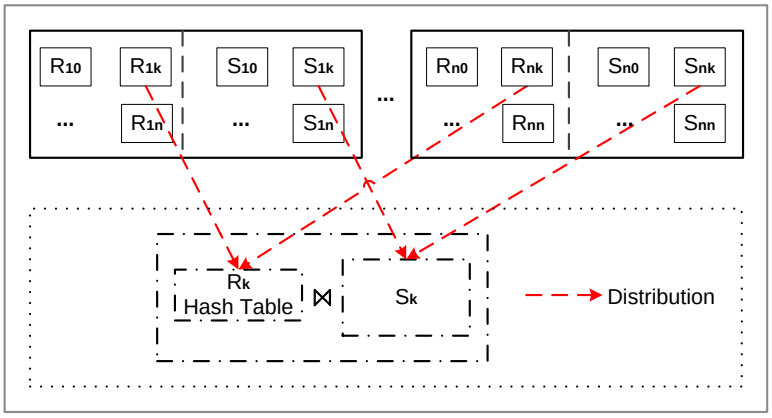


Fig. 2.9 The hash-based distributed join approach. The dashed square refers to the remote computation nodes and objects.

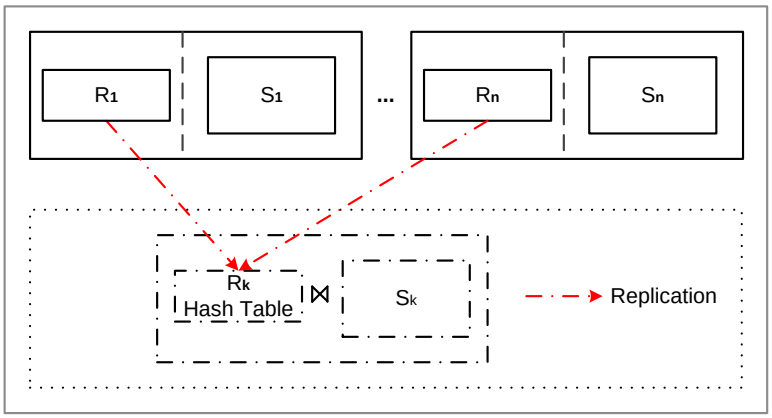


Fig. 2.10 Duplication-based distributed join framework.

In the hash-based framework, the basic parallel join algorithm contains four phases, as illustrated in Figure 2.9: *partition*, *distribution*, *build* and *probe*. In the first phase, the initially partitioned relation R_i and S_i at each node are partitioned into distinct sets R_{ik} and S_{ik} respectively, according to the hash values of their join key attributes. Each of these sets is then distributed to a corresponding remote node in the second phase. These two phases can be considered as a redistribution process, after which, the sequential join of local fragments commence. In the build phase, the relation R_k composed from the redistribution at each node (namely $R_k = \bigcup_{i=1}^{n} R_{ik}$) will be scanned, and an in-memory hash table will be created with the join key attribute. The final probe phase scans each tuple in S_k ($S_k = \bigcup_{i=1}^{n} S_{ik}$) to check whether the join key is in the hash table, and the output will be created in the case of a match.

The duplication-based distributed join framework is shown in Figure 2.10. The join

implementation includes three phases: *duplication*, *build* and *probe*. The first phase just simply duplicates (broadcasts) the tuples of R_i at each node to all other nodes. This means that, after the broadcast, the composed relation R_k at each node will be equal to the full input R, namely, $R_k = \bigcup_{i=1}^{n} R_i = R$. The following two phases are very similar to the final two phases of the hash-based implementation, i.e. that local lookups for S_k will commerce once the in-memory hash table of R_k is created.

Since each phase can be parallelized across nodes, both the schemes above offer the potential for scalability. However there are significant performance issues with both approaches. For the hash-based scheme, while a near linear speedup has been demonstrated under ideal balancing conditions [39] the presence of significant data skew dramatically impacts performance [40] due to node hot spots. Although duplication-based methods can handle skew, the broadcasts of each R_i to all the nodes incurs a heavy time-cost and building a large hash table based on $\bigcup_{i=1}^{n} R_i$ at each node has detrimental impact on performance due to the associated memory and lookup cost [46].

In fact, data skew is a significant problem for multiple communities. For example, databases [75], data management [16], data engineering [20] and web data [78]. Joins with extreme skew can be found in the semantic web field. For example, in [78], the most frequent item in a real-world dataset appeared in 55% of entries. Therefore, it is very important for practical data processing systems to perform efficiently in such contexts.

Dealing with Skew

Currently, different techniques and algorithms have been proposed to handle the join skew [6, 77, 127, 133], and all of them so far rely on the conventional frameworks already described. Further, different techniques, such as DHT [125], dynamic scheduling [81] and statistically based methods [6] etc., have been applied in the implementation of joins to handle the skew issue. Here, we discuss two representative and influential methods - one implements load assignment by *histograms* while the other one is the state-of-art PRPD method. We describe each in turn.

Histograms. Distributed histograms were proposed by Hassan et al. [6] in an effort to improve the redistribution plan to process data skew. Their approach is divided into two parts: (1) histograms for R, S and $R \bowtie S$ are built at each node, in either local or global view or both, and (2) based on the complete knowledge of the distribution and join information of the relations, a redistribution plan to balance the workload for each node is formulated.

Their experimental results show that this method is efficient and scalable in the presence of data skew, nevertheless, two main problems can be identified that hamper its performance:

(1) histograms are built based on the redistribution of all the keys of R and S, which leads to high network communication, and (2) although only the tuples participating in the join are extracted for redistribution, thus reducing part of the network communication, this operation is based on the pre-join of the distributed keys, which incurs a significant time cost.

PRPD. Xu et al. [127] propose a hybrid distributed geography called PRPD (*partial redistribution & partial duplication*) for inner joins, by combining the two conventional patterns described. For a single skew relation S (assume R is uniformly distributed), the high skew tuples S_{loc} of S are retained locally and other tuples S_{redis} are redistributed based on hashing. For R, the tuples R_{dup} with keys contained in S_{loc} are broadcast to all the nodes, and the rest R_{redis} are redistributed as normal. The final joins are composed by $R_{redis} \bowtie S_{redis}$ and $R_{dup} \bowtie S_{loc}$ at each node.

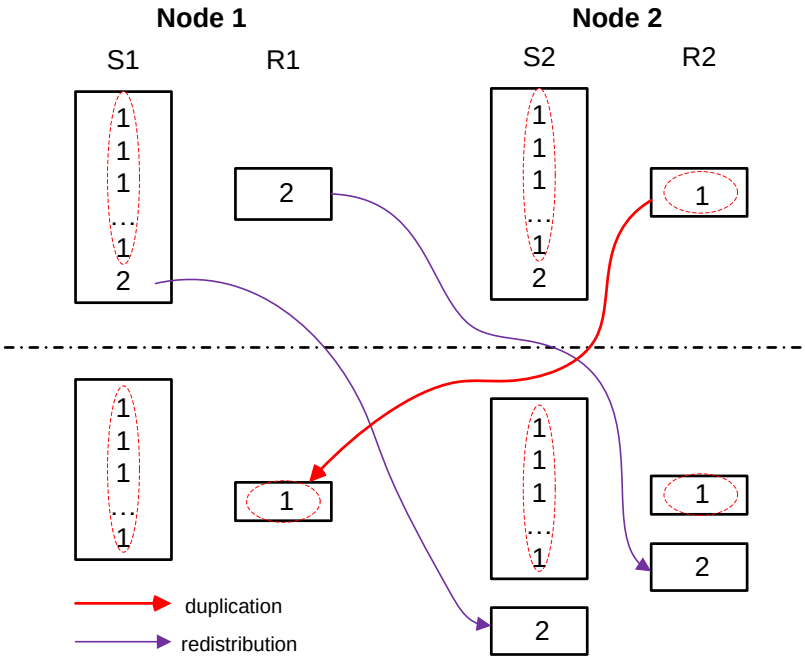


Fig. 2.11 An example of the data movements in PRPD implementation.

An example about the PRPD implementation is demonstrated in Figure 2.11 (only with key operations). The key $\{1\}$ is highly repetitive in the relation S and thus can be considered as skew. For a common hash-based implementations, all the tuples with the key $\{1\}$ in $S2$ will be transferred to the first node and thus lead the first node to be the hot spot. In contrast, in PRPD, none of such tuples will be transferred. Instead, only a single tuple with key $\{1\}$ in $R2$ is duplicated to the first node. In this scenarios, the cost network communication can be highly reduced. Meanwhile, from the final to-be-joined data at each node, it can be seen that the workload at each node becomes more balanced as well.

The experimental results of the PRPD show that this approach can indeed achieve significant performance when compared with the basic hash approach, in the presence of data skew. Even so, PRPD may still suffer from two major problems.

- *Global skew:* Global operations like statistical calculations or broadcasts for the skew keys at each node are required initially. As the split of R and S fully relies on the skew keys in S, the final join will fail if any node does not have global knowledge of such keys.

- *Broadcast:* The processing of the tuples of the duplicated part from R at each node involves broadcasting, which leads to significant network communication as the number of such tuples as well as the number of nodes increases. In addition, it could also bring in redundant join operations.

Consider another similar example in Table 2.1, where the tuples in relation R and S at each node in a 2-node system are shown. Assuming that {1,2} is the skew key set at node 1 while {1,4} for node 2, then the global skew set should be {1,2,4}. If S_2 or R_2 or both relations at node 2 are only partitioned based on a subset of global knowledge, such as the local skew {1,4}, then: (a) in the first case, as $R_{dis} \bowtie S_{dis} = \{3\}$, the join results for the tuples with key {2} in S_2 are lost, and (b) in the last two cases, the join $R_{dup} \bowtie S_{loc}$ only commences on the key {1,4}, and misses the part of {2}. We can also observe that when the relations at node 2 (see $2'$) are partitioned over {1,2,4}, then there is no miss for the output results. In the more general case, we assume that the high skew keys in S_i at each node are simply broadcast so that all nodes can exchange their local skews.

Table 2.1 An example of data partitioning in the PRPD algorithm

n	S	R	**skew**	S_{loc}	S_{dis}	R_{dup}	R_{dis}
1	1,1,1,2,2,3	1,3	1,2,4	1,2	3	1	3
2	1,1,1,2,4,4	2,4	1,4	1,4	2	2,4	$\emptyset$
				1,2,4	$\emptyset$	4	2
				1,4	2	4	2
$2'$	1,1,1,2,4,4	2,4	1,2,4	1,2,4	$\emptyset$	2,4	$\emptyset$

For the second potential issue, following the same example, the duplication part of R is {1} at node 1 and {2,4} at node $2'$, and there exists redundancy in the final join $R_{dup} \bowtie S_{loc\_1}$ over the key {4} at node 1. This redundancy is the result of the uneven partitioning of the

skew tuples in S over each node before the join. In our case, the tuples with key $\{4\}$ of S appear twice on node $2'$ but do not appear on node 1. Because this key is considered as a skew key, all the tuples with $\{4\}$ in R are duplicated and join with the S_{loc_i} at each node, even though S_{loc_i} does not contain $\{4\}$ (e.g. node 1). Obviously, when the number of such uneven tuples is large, the redundant computation will be significant, and will have a severe impact on performance. To improve performance here, [127] proposes a solution that redistributes the skew tuples evenly to all the nodes before the join. However, this pre-redistribution will generate extra communication costs, while more complex and careful global statistical operations for all tuples of S are required. The authors in [127] do not provide any detailed implementation or experimental details regarding this pre-procesing. In contrast, as we will present in Chapter 6, our proposed algorithms does not require such operations at all and can also outperform PRPD under various join workloads.

2.5.2 Outer Joins

Outer joins are popular in complex queries and frequently used in OLAP [47, 95] and large-scale data analysis. Unlike inner joins, the operation does not discard tuples from either relation that do not match with tuples in the other [14]. For example, for a left outer join ($\ltimes$) between two inputs R and S on their attributes a and b, the following `Query 3` returns not only the matched tuples in the form of <x,a,y>, but also <x,a,null>, when values do not match.

```
select R.x R.a S.y
from R left outer join S   on R.a = S.b     (Query 3)
```

Basic Methods

Currently, similar as for inner joins, implementations for distributed outer joins utilise one of two distributed patterns [126]: hash-based and duplication-based outer joins. As *left outer joins* are the most commonly used outer joins, we simply focus on this kind of operation between two relations R and S on an n-node system in the following.

As shown as Figure 2.12, for hash-based approaches, parallel outer joins contain three phases: *partition*, *redistribution* and *local outer joins*. In the first phase, the relations R_i and S_i, initially arbitrarily partitioned across each computation node i, are partitioned into distinct sets R_{ik} and S_{ik} ($k \in [1,n]$) respectively, according to the hash values of their join key attributes. Each of these sets is then distributed to a corresponding remote node k in the second phase. After that, the sequential outer joins of local fragments commence.

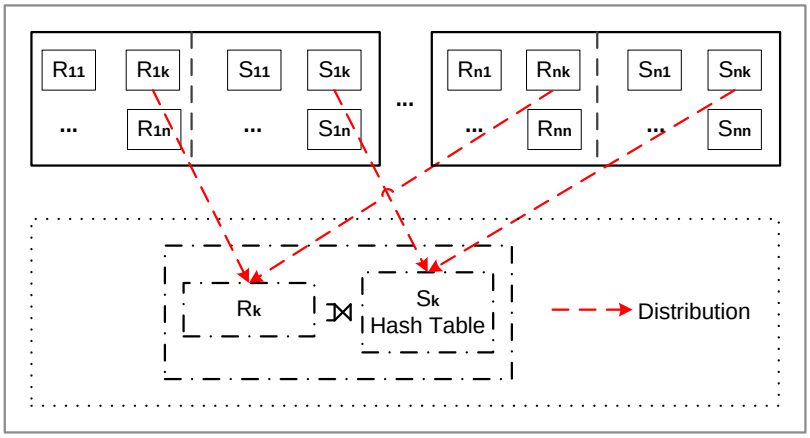


Fig. 2.12 Hash-based distributed outer joins.

Similar to the inner joins, this scheme can achieve near linear speed-up under ideal balancing conditions for distributed systems [39]. However, when the processed data has significant *attribute value skew*, the join performance will dramatically decrease due to the emergence of computational hot spots [40].

Duplication-based outer joins differ significantly from inner joins. As demonstrated in Figure 2.13, there are two distinct stages involved: (1) An inner join between R and S, composed by a *duplication* and *local inner join* phase in which the former phase duplicates R_i at each node to all other nodes, and the latter is the same as that for sequential inner joins, formulating the intermediate results T_i at each node i. (2) An outer join between R and T. This stage is similar to the redistribution-based method described above, but only redistributes T instead. The *duplication* in this framework can efficiently reduce hot spots resulting from *attribute value skew*. Nevertheless, this operation is costly and only suitable for small-large table outer joins. Additionally, such a scheme will still encounter performance bottlenecks when there exists *join product skew* [6], because in such scenarios the redistributed T could be very large (e.g. Cartesian product) or suffer from skew itself.

As stated previously, various techniques have been proposed for distributed inner joins to handle skew [6, 27, 77, 127, 132], regardless, little research has been done on outer joins. The reason for this may be the assumption that inner join techniques can be simply applied to outer joins, as identified in [126]. However, as shown in our evaluations later in Chapter 6, applying such techniques for outer joins directly may lead to poor performance.

Although many systems can convert outer joins to inner joins [49], providing an opportunity then to use inner join techniques, this approach necessitates rewriting mechanisms, which may prove complex and costly. Additionally, current research on outer joins focuses on outer join reordering, elimination and view matching [14, 60, 80]. State-of-the-art

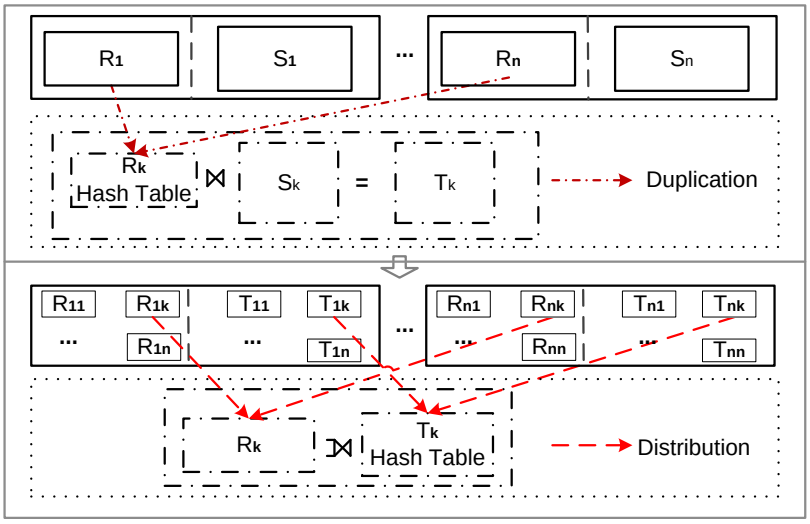


Fig. 2.13 Duplication-based distributed outer joins.

methods designed specified for outer join implementation achieve significant performance improvements [126], however, as we will describe below, they are based on the duplication-based method that can be only applied for small-large table outer joins, which does not meet our requirements in aspect of large-large table outer joins.

State-of-the-art Approaches

Applying PRPD. The described PRPD algorithm illustrates an efficient way to process the high skew tuples. PRPD is actually a hybrid method combining both the hash-based and duplication-based join schemes, in which the two distributed patterns are supported by outer join implementations. Therefore, we can simply use the $R_{redis} \bowtie S_{redis}$ and $R_{dup} \bowtie S_{loc}$ to replace the corresponding inner joins in the scenarios of outer joins. Regardless, there would exist one possible performance issue: the cardinality of the intermediate results in $R_{dup} \bowtie S_{loc}$ will be large because the S_{loc} here is high skewed, and this will bring in significant time-costs.

DER. Xu et al. [126] propose another algorithm called DER (*duplication and efficient redistribution*), which is the state-of-the-art method for optimization of outer joins. The method comprises two stages. (1) They duplicate R_i to all the nodes and then implement the inner joins. In contrast to the conventional approach, they record the ids of all non-matched rows of R at this stage. (2) They do not redistribute any tuples in the second stage, instead, they just redistribute the recorded ids according to their hash values and then simply organize the non-match join results on that basis. The final output is the union of the inner join results in

the first stage and the non-matched ones in the second stage.

In fact, this method shows an efficient way to extract non-matched results. Notice that the *join* in the first stage of the conventional duplication-based method is a inner join but not a outer join, the reason is that the outer join brings either redundant or erroneous non-matched output. For a two-node system for example, if the output of the duplicated tuple {1,a} is {1,a,null} on both nodes, there is no match for this tuple in S and there is a redundant output. Further, if the {1,a,null} appears only on one node, there is a match on the other node, output {1,a,null} will result in error. The conventional approach to alleviate this problem is by redistributing the intermediate results. We can also use another, naive, way to solve this problem by outputting the non-matched results and then redistribute them. Regardless, DER uses a more ingenious way, in that each tuple can be indicated by a row-id from the table R, which is redistributed. Consequently, the network communication and the workload can be greatly reduced, and their experimental results demonstrate that the DER algorithm can achieve significant speedups over competing methods.

As DER must broadcast R_i, it is designed to work best for small-large table outer joins. In this scenario, since R is small, the redistributed part in the second stage will remain small even when S is skew. This is because DER only processes the non-matched part, the number of which is always less than $|R|$ at each node. In contrast with the PRPD algorithm, the broadcast part R_{dup} is typically small, and we expect that integrating DER into PRPD can fix the skew problem as described for $R_{dup} \rtimes S_{loc}$ previously. Our experiments in the Chapter 6 will demonstrate that that this hybrid method (which refer to as PRPD+DER) is indeed very efficient on handling skew in large-large outer joins. Regardless, this approach inherits the two performance issues of PRPD as described, and we will also show that our proposed algorithms can still outperform this optimized technique during join implementations.

2.6 X10 Parallel Programming Language

X10 [22] is a multi-paradigm programming language developed by IBM. It supports the asynchronous partitioned global address space (APGAS) model and is specifically designed to increase programmer productivity, while being amenable to programming shared memory and distributed memory supercomputers.

It uses the concepts of `place` and `activity` as the kernel notions to exploit parallelism in the available hardware. A place is a logical abstraction of the underlying heterogeneous processing element in the hardware such as cores in a multi-core architecture, GPUs, or a whole physical machine. Activities are light-weight threads that run on places. X10 sched-

ules activities on places to best utilize the available parallelism. The number of places is constant through the life-time of an X10 program and is initialized at program startup. Activities on the other hand can be forked at program execution time. Forking an activity can be blocking, where in the parent returns after the forked activity completes execution, or non-blocking, where in the parent returns instantaneously, after forking an activity. Furthermore, these activities can be forked locally or on a remote place.

X10 provides an important data structure called distributed arrays (`DistArray`) for programming parallel algorithms. One or more elements in the `DistArray` can be mapped to a single place using the concept of points [22]. Additionally, we used the following three crucial parallel programming constructs for our compression implementation.

- `at(p) S`: this construct executes statement `S` at a specific place `p`. The current activity is blocked until `S` finishes executing on `p`.

- `async S`: a child activity is forked by this construct. The current activity returns immediately (non-blocking) after forking `S`.

- `finish S`: this construct is used to block the current activity and then waiting for all activities forked by `S` to terminate.

Based on the experiences derived from the development work of this thesis, we find that there are a number of advantages to using the X10 language, and in turn the APGAS model, to implement parallel/distributed algorithms/systems: (1) flexible and efficient scheduling. APGAS, like PGAS, separates tasks from the underlying concurrency model, thereby allowing one to implement an efficient scheduling strategy irrespective of the number of tasks forked using `async`; (2) APGAS, being derived from both MPI and OpenMP programming models, extracts parallelism at both the distributed and single machine hierarchies; and (3) the abstract programming model supports the development of succinct code which is easier to debug and maintain.

2.7 Conclusion

In this section, we primarily presented related work to that of this thesis, focusing on current high performance RDF stores, RDF benchmarks and associated techniques such as indexing, data encoding and parallel joins.

In the first two sections, we introduced the novel data structure and optimization techniques employed in some typical single-node triple stores, which can compute queries very

fast. We also classify the indexing methods used in current distributed solutions into four types and show that there exists complexity trade-off between them, in terms of data loading and querying performance. Then, we describe the existing RDF benchmarks as well as four popularly used benchmark datasets. Additionally, we present related evaluation work on RDF stores.

In the latter three sections, we first presented current data compression techniques, especially the two distributed dictionary encoding implementations, one implemented on a Cray XMT supercomputer and one based on the MapReduce model. After that, we described the detailed implementations of different parallel inner- and outer joins. We focused on analyzing the performance issues of the conventional join approaches while demonstrating how state-of-art methods can efficiently handle data skew over large scale data. Finally, we introduced the modern parallel programming language X10 highlighting its advantages for development, from the basis of our own development experiences.

Chapter 3

Runtime Characterization of Triple Stores

3.1 Introduction

The performance of RDF stores becomes increasingly important with the growth of the Semantic Web. As described in Chapter 2, current RDF benchmarks and experiments are concentrated on evaluating the response time and query throughput of individual stores to show the general weaknesses and strengths of RDF implementations, but have not sufficiently given insight reasons for their conclusions.

In this chapter, we focus on a more detailed analysis of RDF stores. Especially, we try to conduct a more detailed system-level evaluation of currently triple stores, with the following three main targets:

- To allow the dynamics and behaviors of RDF query execution to be better understood.
- To discover performance inhibitors and bottlenecks of current RDF stores.
- To give insightful suggestions for RDF store developments and help in the design of efficient distributed stores, optimized for parallel RDF processing.

We choose four of the most popular and mature triple stores, available as open-source software: Jena [86] and Sesame [19], both written in Java, RDF-3X [89], a state-of-the-art store for scalable SPARQL processing and, Virtuoso [44], a commercial multi-purpose and multi-protocol data store. Compared to the conventional high-level evaluations, we construct several new suitable metrics for RDF stores, focus on profiling their low-level

implementations. Moreover, we create triples up to 5 billions on the basis of the BSBM [15] benchmark and implement our experiments over two different platforms: a standard (128GB RAM, 12cores, standard HDD) and an enterprise platform (768GB RAM, 40 cores, enterprise SAN).

Rather than a common benchmarking effort, the work present in this chapter should be read as an analysis of the runtime Characterization of queries for a representative set of RDF stores. We only consider loading times with regard to the feasibility of our experiments and we do not focus on the compliance to the SPARQL specification or the feature-set of each store. In this light, our measurements and analysis aim at guiding development of RDF stores, rather than evaluating existing ones.

The rest of this chapter is structured as follows: In Section 3.2, we provide a general work flow of RDF query processing in triple stores. The detailed methods we use to collect the data for our proposed metrics are shown in Section 3.3. In Section 3.4, we describe the experimental environments. We present the test results and discussion in Section 3.5 and conclude in Section 3.6.

3.2 RDF Store Querying

Query performance is always the most important issue for an RDF system, therefore we focus on the query process in this section.

Similar to traditional database systems in terms of system architecture, the main components of Jena, Sesame, RDF-3X and Virtuoso include a query engine, a storage subsystem and a database. The query engine is used to parse the query from a user or an application program and produce an execution plan, represented as a tree of relational operations [70]. The storage subsystem includes a buffer or even its own file cache manager, which, as the name suggests, manages the buffering of data and reduces the number of disk accesses.

The general process of query implementations for the four triple stores is illustrated in Figure 3.1, which comprises three main phases from top to bottom: *query parsing*, *query planning* and *query execution*. Unlike performance evaluations done previously, which have only focused on the time cost of entire query process, in our evaluations, we will measure the time cost of each phase to track performance more precisely.

To better understand the insight implementation of triple stores, we examine the detailed process of the three phases in turn. As the first query parsing process is simple, namely the strings of input queries are analysed based on the SPARQL syntax, here, we just focus on the latter two phases.

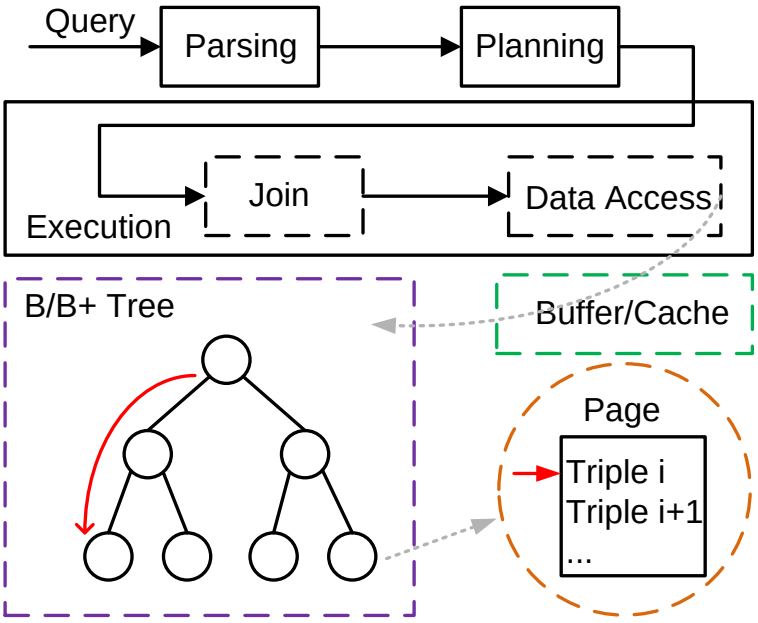


Fig. 3.1 The work flow of the general query process in triple stores.

3.2.1 Query Planning

The sequence of underlying implementations of a query such as joins rely on the responsible query plan, and an unoptimized plan will bring large number of redundant intermediate results and thus impacts the query performance because of the memory consumption as well as result materialization. SPARQL queries typically generate deep query plans and RDF lacks information about access patterns available in relational databases (e.g. foreign keys). This makes query planning, and in particular join order optimization, challenging and resource-consuming. Currently, most RDF store's query optimizer can only collect limited statistics, such as RDF-3X's histograms as described previously, and Jena which collects the number of times a predicate appears. Further, some systems (for example, Virtuoso) cache query plans for later use. We will examine this time for each store so as to demonstrate their differences.

3.2.2 Query Execution

Joins. The join implementations in an RDF store has been extensively studied in previous chapters - candidate results of two sub-graphs will join based on their join keys following the responsible query plan. As the execution time of SPARQL queries is dominated by such operations, here we just report on the join methods used in the four stores examined.

Namely, Jena and Sesame only use nested-loop joins, RDF-3X uses merge joins as well as hash joins and Virtuoso uses all the three types of joins. The latter two systems always choose the most efficient joins in the planning phase according to the cost of each kind of join, which means that they could spend more time on the query planning and consequently reduce the query execution cost.

Data Access. If a join operation organizes the general operation of all the triple patterns in a query, then the data access process can be considered as the detailed implementation of retrieving bindings for single triple patterns. This process is always costly and thus an efficient indexing structure is always needed so as to enable fast location of the required data pages and then retrieve them. Jena [86], Sesame [19] and RDF-3X [89] use B/B+ Tree indexes, suitable for range queries, and the index scheme of Virtuoso contains primary key and bitmap indices. Jena also provides three triple indexes on *spo*, *pos* and *osp* to accomodate different triple patterns, while Sesame offers two indexes *spoc* and *posc* by default, and RDF-3X maintains 15 indexes (6 indexes and 9 aggregated indices) for covering all the possible join patterns. The redundancy is offset by index compression methods. Virtuoso provides two full indexes *posg* and *pogs*, where the *g* indicates the *graph name*, and three partial indexes *sp*, *op* and *gs* as default. All systems use a dictionary, mapping values to numeric identifiers. The triple indexes, and most operations, operate on these numeric identifiers.

Data Caches. Practically all RDF stores (and all databases) employ caching mechanisms for triple indexes and dictionaries to improve the performance of frequently encountered queries. This kind of data cache is always implementation-specific. For example, Sesame employs a caching and buffering approach using the Java heap. During data retrieval, it will access the buffer or cache to check whether the required data is there and start an index scan. If there is no matched data, the needed B-tree node will be read into the buffer first before seeking to the exact data position. Depending on the location of the requested data, some B-tree nodes will be processed directly, some will be read from the disk cache and some will be read directly from disk. Caches influence data access operations like *index scans*, *page reads* and *triple lookups*. To obtain a more precise description of the performance of such operations, we record the number of index scans and their timing, the number of the pages read and the number of the triple lookups for a single query. All these data is useful for describing the dynamics of data searching, which is directly associated with query performance.

3.3 Methodology and Metrics

The previous section gives insight to the workings of RDF stores in general. In this section, we describe in detail the methodology and the metrics used in our experiments. We have instrumented Jena, Sesame and RDF-3X by modifying their source code. Virtuoso already provides some metrics[1] itself, which we retrieve using the Virtuoso JDBC driver.

We measure the time cost for the parsing phase starting from the time we get a query string (in-process), to the time we get the query tree. At this point, the planning phase starts. We consider that the planning phase is finished when we get an execution plan. Note that for the purposes of this chapter, any runtime decisions (for example, sideways information passing techniques used in RDF-3X) are counted as part of the execution phase and not as part of the planning. The execution phase is finished when the last result has been received.

```
   Index and Search Range [a,b] have been confirmed.
 1: Counter1: scan_start_time
 2: read (index.root())
 3: binary search to get child node
 4: while triple_id < a do
 5:       read (child.node())
 6:       Counter2: page_read_1++
 7: end while
 8: release(tree.root())
 9: end read
10: Counter3: scan_number++
11: Connter4: scan_end_time
```

Fig. 3.2 Pseudo codes of four counters in a scan implementation.

For Virtuoso, we retrieve other relevant metrics using the corresponding SQL statements after each query. For the other three systems, all other metrics are collected through inserting counters in their program codes. For example, four counters are assigned for an index scan as pseudo codes are demonstrated in Figure 3.2. We define the start of an index scan with the reading of the root node of the index and the end with release of the root node. A separate counter for the number of pages accessed is used in this process (`Counter 2` in the code), here it indicates only part of pages read, while there would be no scan when reading the consequent content pages, where we also insert the `Counter 2`.

[1] http://docs.openlinksw.com/virtuoso/ptune.html

A triple is located in a slot of data page and an extra id is used to indicate its slot. Our lookup counters increment every time when the system looks up this id to check whether it meets the searching range. For RDF-3X, the triple lookup operation happens during its scan process. For Jena and Sesame, the lookups are accompanied with the results retrieve process.

Lastly, we monitor the CPU usage during all the tests with SYSTEMTAP[2], a tool for gathering information about system utilization in the Linux operating system. The CPU was sampled every second. For the purposes of this work, we consider 100% CPU usage when a single logical processing unit is fully utilized (i.e. a dual-core machine with two threads per core can have up to 400% CPU usage).

We will present results for some metrics across all queries and for some others, we will focus on specific queries. The metrics used here are summarized in Table 3.1, where the *general* means the most common used metrics and the *detailed* indicates the new ones we proposed.

Table 3.1 Metrics List

Metrics	General	Detailed
Data Loading Time	✓	
Disk Space Consumption	✓	
QMpH	✓	
Query Parsing Time		✓
Query Planning Time		✓
Query Execution Time		✓
Number of Index Scans		✓
Scan Time		✓
Number of Lookups		✓
Number of Read in Pages		✓

3.4 Experimental Settings

3.4.1 Benchmark

For our experiments, we have used the Berlin SPARQL Benchmark (BSBM) [15]. BSBM generated synthetic datasets of arbitrary size, representing an e-commerce use-case in which

[2]http://sourceware.org/systemtap/

a set of products is provided by various vendors and consumers post reviews around those products. We created a series of datasets, the largest of which is composed of 5 billion triples, occupying around 1.2 TB in N-Triples format.

In terms of queries, we have concentrated on the explored use-case of BSBM. Although the corresponding query set is suitable for use with Jena, Sesame and Virtuoso, several query features such as the aggregation operations *describe* and *optional* are not supported by RDF-3X. Consequently, we rewrote the queries[3] to cater to RDF-3X.

3.4.2 Platform

All the experiments were conducted on two platforms, a *standard platform* (SP) and *enterprise platform* (EP). Their configurations are shown in Table 3.2. The standard platform we have used is an iDataPlex node with 2 Intel Xeon X5679 processors, 128GB RAM with a single 1TB SATA HDD. The enterprise platform consisted of a high-memory server IBM x3850 X5, an enterprise-grade IBM XIV SAN and two IBM System Storage SAN48B-5 fiber channel switches. The server was equipped with 4 Intel Xeon E7-8850 processors, 768GB RAM and two Emulex 8GB FC Single-port HBAs, each connected to one switch. The XIV SAN used 156 HDDs and was connected to each switch on six fiber channel ports.

Table 3.2 The Configurations of Test Platforms

Machine	Standard Platform	Enterprise Platform
CPU	2*6 Cores, 2.93GHz	4*10 Cores, 2.00GHz
RAM	128GB	768GB
Disk	1TB	XIV SAN
Linux Kernel	rhel-2.6.32-220	rhel-2.6.18-308
Java Version	1.6.0_25	1.6.0_25

3.4.3 Setup

We have experimented on Jena v2.6.4, Sesame v2.6.5, RDF-3X v0.3.7 and Virtuoso Open Source v6.1.5. We set the Java heap size for Jena and Sesame to 40GB on SP and 240GB on EP. For Virtuoso, the system parameters *NumberOfBuffers* and *MaxDirtyBuffers* were set to 5242880 and 3932160 on SP and 31457280 and 23592960 on EP. For Jena, we have configured the optimizer with the statistic optimization strategy. For Sesame we have set

[3] http://code.google.com/p/para-computing-long/downloads/list

the index configuration to *spoc*, *posc* and *opsc*. The rest of the parameters were left to the default values.

Moreover, we chose 150 query mixes for our experiments, of which 50 query mixes were used in the warm-up phase and the other 100 were in the hot-run. To minimize the caching effects of previous queries, we empty the file system cache before running the query mixes of each test. Additionally, our experiments have been limited by the following three conditions, in terms of systems and time limitations: (1) hard disk space, (2) loading time (with a cut-off at 100 hours) and (3) query execution time (with a cut-off at 24 hours).

3.5 Results and Discussion

In this section we present and analyze the results derived from the metrics described previously and provide insight on the runtime characteristics of the aforementioned RDF stores for the platforms used.

3.5.1 Loading

The results of loading time and disk consumption over SP and EP are shown in Figure 3.3 and Figure 3.4 respectively. Sesame performs poorest in terms of loading capability - 250M triples take more than 100 hours. A possible explanation for this lies in the relatively small page size used by Sesame, which is only 2KB, that leads to very frequent index updates. On EP, the situation is improved as 500M is loaded in about 34 hours. For Jena, 500M was loaded in 80 hours for SP while the performance on EP was dramatically superior - 5B loaded in 70 hours (not shown). In comparison, RDF-3X took 75 hours. Virtuoso and RDF-3X achieved much faster loading speeds than Jena and Sesame on SP, but their loading time on EP is at the same levels as in SP, indicating that they did not exploit the hardware. In the meantime, it takes days for the four systems to load large datasets, highlighting the requirements for parallel processing in the presence of big RDF data.

For disk space requirement, it can be observed that this metric is linear increasing with the increment of loaded triples. Moreover, the index compression methods of RDF-3X pay off, resulting in the smallest index size. Virtuoso (also using index compression) uses nearly 10% more space. Jena and Sesame generate much larger indexes about 2 times larger than those of RDF-3X, though they have less indexes.

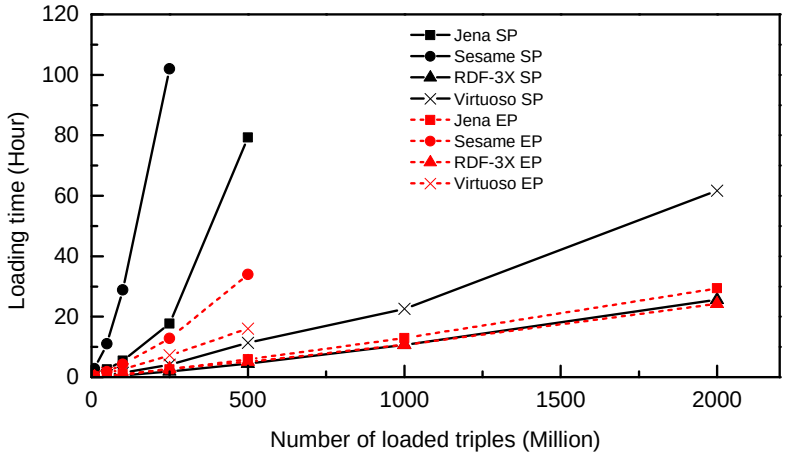


Fig. 3.3 Data loading time on the two platforms.

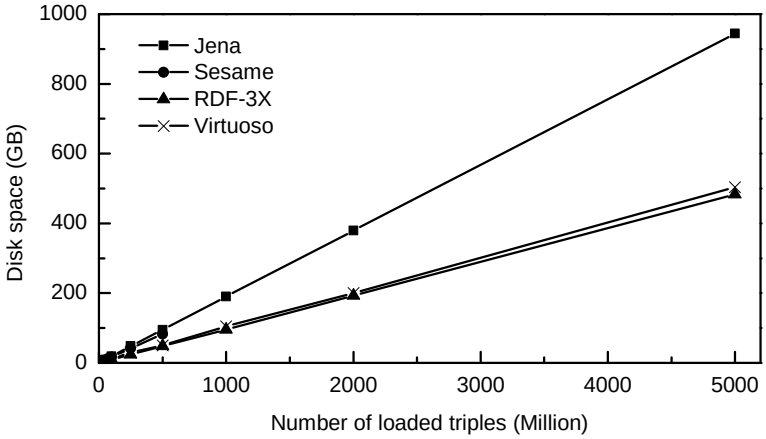


Fig. 3.4 Disk space required for various datasets.

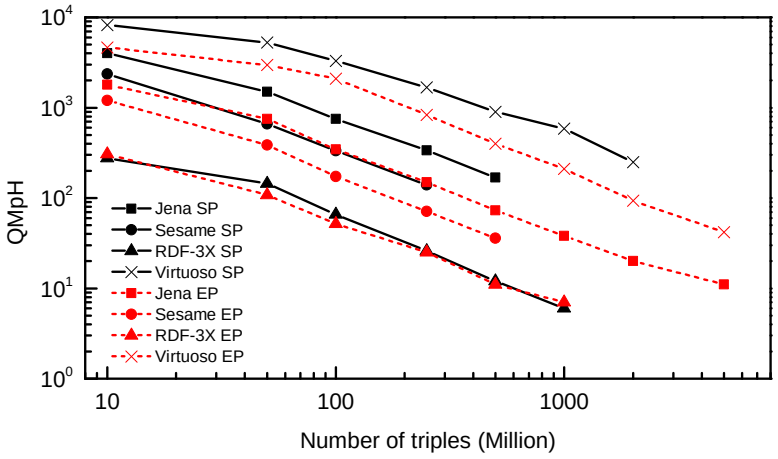


Fig. 3.5 QMpH for various datasets on the two platforms.

3.5.2 QMpH

There are 25 queries in a Query Mix, which is the same as the BSBM configuration, and Figure 3.5 shows the QMpH result on the basis of our rewritten queries. As expected, performance decreases with an increasing dataset size for all stores in both platforms. This change is essentially linear as demonstrated in the figure. We also note that RDF-3X performed the worst of all stores, which comes in contradiction with the description in [90]. We will explain this further in the following part about the cost breakdown of query implementations.

Comparing the two platforms, EP appears to have worse QMpH than SP, which is surprising. In terms of hardware configuration, SP has only one advantage, namely CPU clock speed. We draw the conservative conclusion that computation is CPU-bound and explain this further in Section 3.5.7.

With the results mentioned above, regarding to our test strategy, the maximum number of triples for each store in our experiments is also expressed in Figure 3.5 according to the terminal points of different curves. We measured the QpS of 250M triples on SP for all stores, which we believe could indicate a general performance measure for our test. Based on that, we also listed three queries with the best QpS and three with the worst for each store as shown in Table 3.3. Since Q5, Q8 and Q12 appear frequency in that list, with the interest in outstanding queries, we chose these three queries as main analysis objects for our proposed metrics in the following.

Table 3.3 Special queries for RDF stores with 250M triples on standard platform

RDF Store	Best Queries	Worst Queries
Jena	Q2, Q9, Q12	Q3, Q4, Q5
Sesame	Q2, Q9, Q12	Q5, Q10, Q11
RDF-3X	Q2, Q11,Q12	Q5, Q7, Q8
Virtuoso	Q2, Q9, Q12	Q3, Q5, Q8

3.5.3 Cost Breakdown

Table 3.4 shows the cost breakdown between query parsing, planning and execution, across all stores and queries for 250M triples. For most stores, the runtime is dominated by execution time. Query parsing represents a small fraction of the cost, so we will exclude it from further discussion. Planning cost differs significantly per store, with Virtuoso spending significantly more time than the other stores. For Q5, Q7 and Q8, we see that the execution

time of RDF-3X is nearly 100%. Especially for Q7, which appears four times in the query mix, and its QpS for RDF-3X is only 0.03, which is extreme low while other stores is in the order of ten, leading RDF-3X a worse QMpH as described.

Table 3.4 Breakdown of different queries for 250M triples on the standard platform (in %)

Q.	Jena			Sesame			RDF-3X			Virtuoso		
	Par.	*Pl.*	*Exe.*	*Par.*	*Pl.*	*Exe.*	*Par.*	*Pl.*	*Exe.*	*Par.*	*Pl.*	*Exe.*
1	2	1	97	4	96	0	0	82	18	0	100	0
2	20	16	64	21	77	2	0	98	1	0	96	3
3	2	1	96	4	96	0	0	87	13	0	100	0
4	2	1	97	4	96	0	0	93	7	0	100	0
5	0	0	100	0	0	100	0	1	99	0	100	0
6	-	-	-	-	-	-	-	-	-	-	-	-
7	3	2	96	2	5	93	0	0	100	0	97	3
8	2	1	97	2	3	95	0	1	99	0	100	0
9	2	2	96	2	96	2	0	100	0	0	96	4
10	3	1	96	1	2	97	0	7	93	0	98	2
11	2	2	96	0	0	100	0	76	24	0	71	29
12	3	2	95	5	94	1	0	90	10	0	97	3

3.5.4 Planning and Execution

Query 12 is chosen as being representative for further analyzing planning costs. The results in Figure 3.6 show that: (1) Planning costs for Virtuoso and Jena are fairly constant and are not significantly influenced by dataset size. We attribute this to the plan caching and the statistical approach taken in these systems respectively. (2) Sesame and RDF-3X clearly demonstrate an increase in planning costs as the dataset size increases. This means that these two systems have more complex optimization strategies in the presence of different workloads.

Among the four stores, the query planner of Virtuoso dominates its query runtime, especially for Q5 (not shown). For this query, the whole query runtime is 808.4 ms and Virtuoso takes 808.3 ms on query planning. This illustrates that performance failures in the query plans can be lethal in RDF stores. On the other hand, the significant effort for optimization pays off, as shown in the execution times in Figure 3.7, where Virtuoso significantly outperforms other stores. We should nevertheless note that this optimization cost is not amortized over the (lower) execution time, as we describe next.

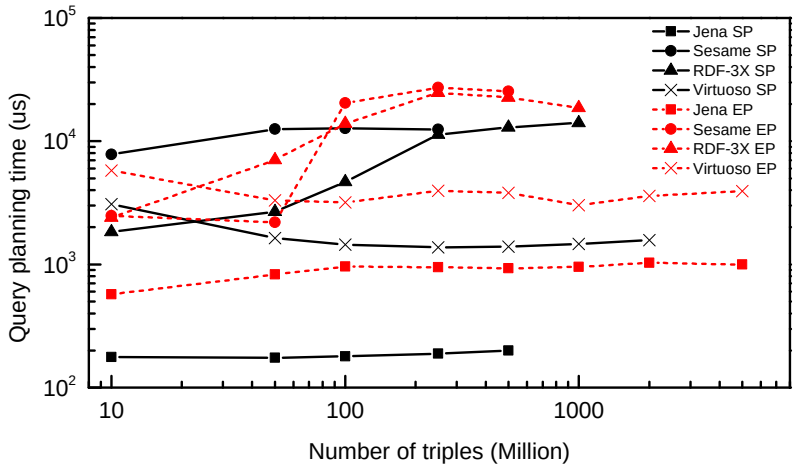


Fig. 3.6 The planning time of Query 12 by varying the number of triples (in logscale).

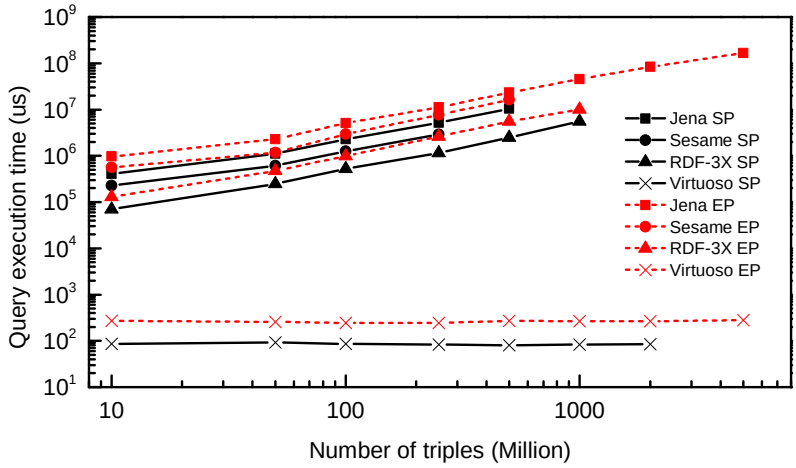


Fig. 3.7 The execution time of Query 5 by varying the number of triples (in logscale).

Query 5 is a bad query for all four stores, as evident in the execution times presented in Figure 3.7. It can be seen that the execution time is basically linear with the data size for Jena, Sesame and RDF-3X, with the latter performing better. The time cost of Virtuoso is practically constant with the dataset size, indicating that a large portion of the computation for this query is done during the planning phase as stated above.

3.5.5 Number of Scans and Scan Time

In terms of number of scans and scan time, the Virtuoso-provided metric *locks* is always 0 for all the queries in our experiments, which indicates no index is locked during the query implementation, we assume the reason is that perhaps the results are stored in the store cache

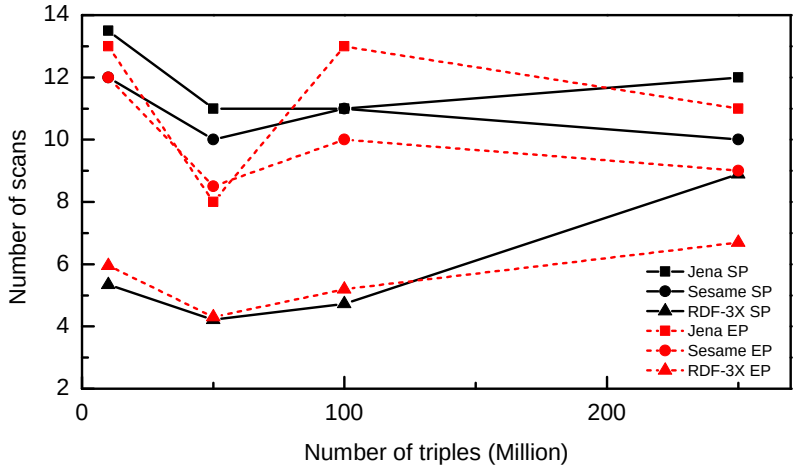


Fig. 3.8 The number of index scans of Query 8 by varying the number of triples.

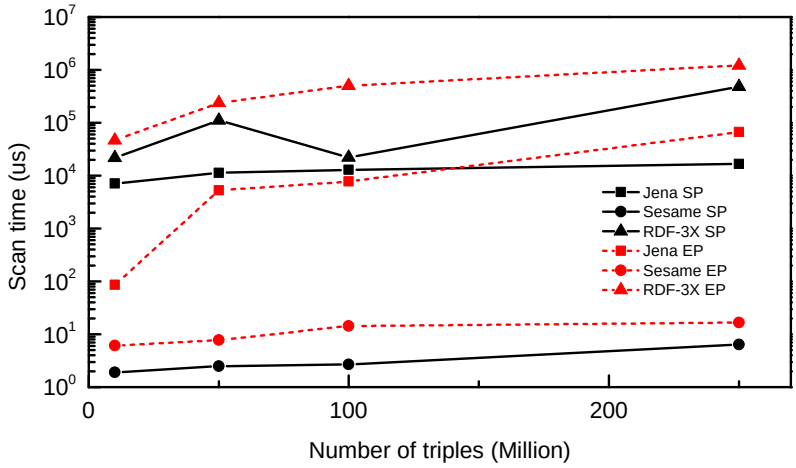


Fig. 3.9 The scan time of Query 8 by varying the number of triples.

and there is no need to search the triples through an index scan. And it is also possible that the lock refers to the number of locks needed for synchronization, and the implementation contains only read operations. We report then *number of scans* and *scan time* only for Jena, Sesame and RDF-3X, since the instrumentation in Virtuoso does not support these metrics. In Figure 3.8 and Figure 3.9, we show results for Query 8 respectively. The curves do not change after 250M triples. It can be seen that the number of scans for RDF-3X is smaller than Jena and Sesame. We attribute this to the fact that RDF-3X maintains indexes for all term permutations. For the time spent on scanning, as shown in Figure 3.9, it is significantly higher for RDF-3X (again, this is due to its architecture). In the same figure, Sesame spent much less time on scanning, since it is using its proprietary in-memory cache, namely part of the required data can be retrieved directly without any scans, compared to the file cache

used by Jena.

3.5.6 Number of Lookups and Read in Pages

Figure 3.10 and Figure 3.11 show the number of triples retrieved (triple lookups) and number of read in pages for Query 5 respectively. These both grow linearly with dataset size. Given the fact that this operation heavily relies on sequential data access. In the meantime, it can be observed that RDF-3X retrieves large number of data on both metrics. In comparison, Jena and Sesame performs nearly the same, are much smaller. The difference in page reads between Sesame and Jena is attributed to the proprietary cache of the former. Comparing these two results with Figure 3.7, we see that even though RDF-3X accesses more data, it strongly outperforms Sesame and Jena, which indicates that the implementation of *joins* in RDF-3X is much faster. This means that the operations of merge-join and hash-join could be potential faster than nested joins in this scenario.

3.5.7 CPU Usage

For all systems, we observed very low CPU usage and although Jena, Sesame and Virtuoso support concurrent evaluation of multiple queries, no system parallelizes the execution of single queries. The CPU usage of Jena and Sesame was almost identical ($70\% \sim 80\%$) on both platforms. RDF-3X and Virtuoso reached 100% CPU on the standard platform, and Virtuoso reached 200% on the enterprise platform. Given that SP and EP have 24 and 80 logical processing units respectively, none of the systems exploit the parallel nature of modern architecture for the evaluation of single queries.

3.6 Conclusions

This chapter has conducted a comparative analysis of the runtime characterization of a representative set of RDF stores, namely, Jena, Sesame, RDF-3X and Virtuoso. We have described the dynamics and behaviors of the query execution on the basis of experimental data and queries derived from the BSBM benchmark.

The main findings of this work are the following: (1) Investing in query optimization pays off in general, but, in SPARQL, it is easy to arrive at a situation in which the runtime performance is dominated by optimization. (2) Planning failures are potentially catastrophic. In our experiments, although RDF-3X was the fastest system in most queries, failure in a single query resulted in it having the worst overall performance. (3) None of the

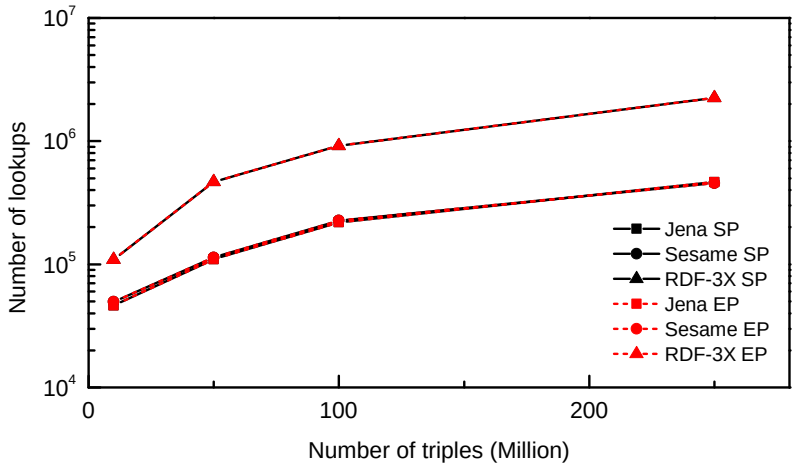


Fig. 3.10 The number of triple lookups of Query 5 by varying the number of triples. For each dataset, Jena and Sesame performs nearly the same, are much smaller than RDF-3X.

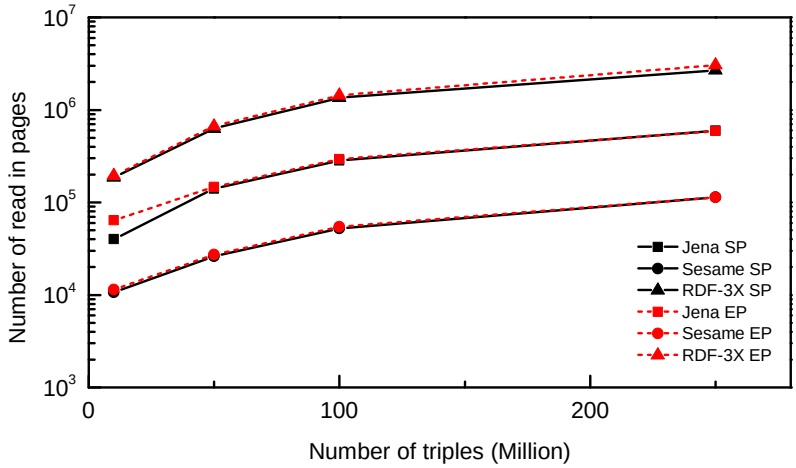


Fig. 3.11 The number of read in pages of Query 5 by varying the number of triples.

RDF stores examined can exploit modern parallel architectures for single queries. This is expected to have a very negative effect on analytical workloads. (4) Using very fast storage, in most cases, did not have the expected impact on performance. This indicates that either the datasets used were completely served by data in memory and caching techniques performed adequately, or that query processing in RDF stores is actually CPU-bound.

All these investigations and findings demonstrate detailed aspects of triple stores and also provide deeper understanding for their behaviors during query execution. This helps us to confirm the main modules and the data flows of our proposed analytical framework as presented in the Figure 1.8 of Chapter 1 in a parallel case. Additionally, the results presented in this chapter show that standalone stores could encounter serious performance bottlenecks

in the presence of big RDF data. Therefore, in the following Chapter 4, we will investigate how to efficiently process large datasets using distributed systems.

Chapter 4

Design and Evaluation of Parallel Hashing over Large-scale Data

4.1 Introduction

The previous chapter has provided a detailed analysis of the runtime characteristics of triple stores. The collected results have also demonstrated that RDF stores meet performance bottlenecks in the face of huge RDF data as the limitation of sequential implementations. We aim to apply parallel techniques to large-scale RDF management systems, namely, we have to find efficient parallel strategies to process big data at first. For example, the detailed parallelism patterns or thread cooperation strategies etc. over a distributed system need to be considered. As hash tables are commonly used in high-performance analytical data processing systems, which often run on servers with large amounts of memory, and they have been employed in the implementations of encoding, joins and indexing of our proposed framework, the focus of this chapter is on investigating efficient parallel hashing algorithms for processing massive data.

In fact, hash tables are the dominant structure for applications that require efficient mappings, such as database indexing, object caching and string interning. The $O(1)$ expected time for most critical operations puts them at a significant advantage to competing methods, especially for large data problems. Regardless, similar to other big data problems, as applications grow in scale, parallel hashing on multiple CPUs and/or machines is becoming increasingly important. Currently, there are two dominant parallel hashing frameworks that are widely used and studied: distributed and thread-level parallel hashing.

For the first framework, as shown in Figure 4.1, the threads at each computation node (either logical or physical) build their own hash tables first, and then process the initial

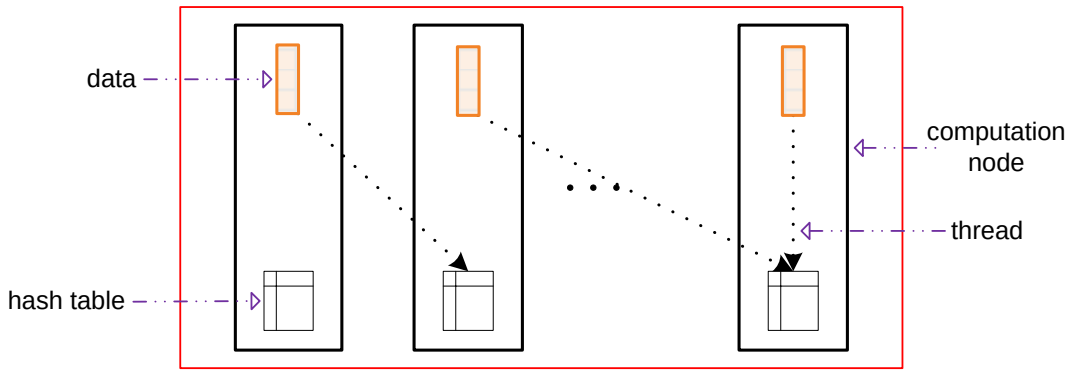


Fig. 4.1 Distributed-level parallelism.

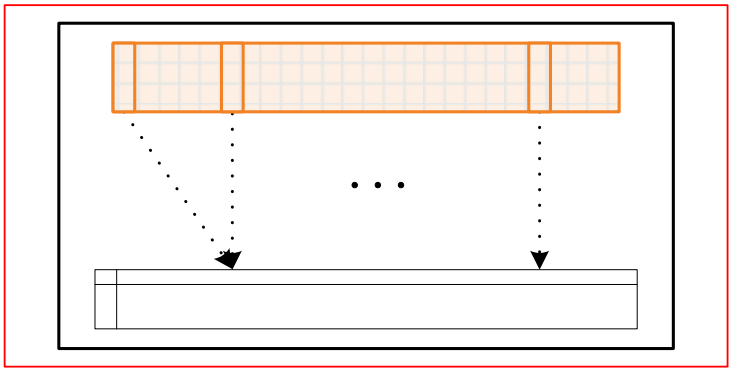


Fig. 4.2 Thread-level parallelism

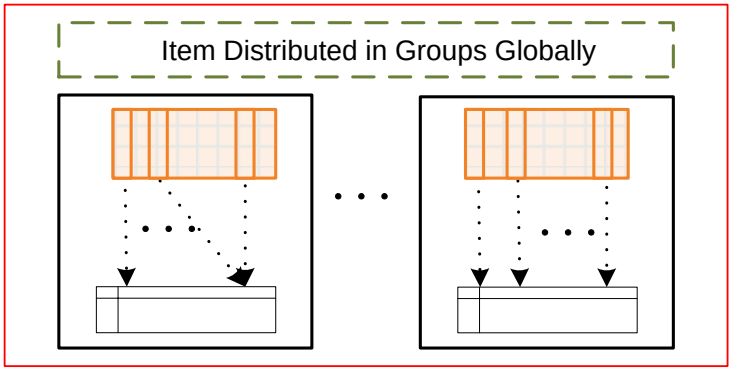


Fig. 4.3 Structured parallelism

partitioned data (refer as *keys* for simplification throughout this chapter) through accessing a local or remote hash table(s). In general, this access is determined by hash values of the processed keys. This approach is very popular in distributed systems. Considering the target for high performance computing, in the following we only discuss the conditions of full parallelism, rather than the hash tables used in *peer-to-peer* systems, for example, the commonly studied *Distributed Hash Tables* (DHTs) [107].

In thread-level hashing, (Figure 4.2), a single hash table is constructed on the single underlying platform, and multiple available threads operate with coordination on that table in parallel. This particular model is widely studied for multithreaded platforms which range in scale from commodity servers to supercomputers. As there exists no costly network communication (though possible NUMA) under this scheme, it always performs very fast.

The two parallel schemes scale in terms of processing large numbers of items by employing new nodes or threads. However, both approaches meet performance issues when processing massive data. With distributed hashing, the large number of frequent and irregular remote accesses of hash operations across computational nodes is costly in terms of communication. Moreover, when the processed data has significant skew, the performance of such parallel implementations will dramatically decrease because all the popular keys will flood into a small number of nodes and cause hot spots. For parallel hashing on multithreaded architecture platforms, the cooperation between threads can efficiently balance the workloads, regardless, both for the skewed or non-skewed data, the associated scalability is bound by the limit on the number of threads available, the availability of specialized hardware predicates and possible memory contention. Furthermore, memory and I/O eventually also become bottlenecks at very large scale.

In general terms, the memory hierarchy of modern clusters consists of a distributed memory level (across nodes) and a shared memory level (multiple hardware threads/cores accessing the memory of a single node). We are proposing a *structured parallel hashing* (SPH) framework (shown in Figure 4.3) that blends distributed hashing and shared-memory hashing, divided into two phases: (1) items are grouped and distributed globally by each thread, and (2) hash tables are constructed on each node and each of them is only accessed by a local thread(s).

The primary idea is a straightforward *bulk-operation* scheme, however, in so far as we are aware the approach has not been previously described in the literature. Intuitively, this method has two advantages: (a) reduced remote memory access, load imbalancing and the associated time-cost arising from memory allocation, table locks and communication in distributed hashing, and (b) support for high scalability compared to thread-level hashing

(there are no hardware limitations as our approach operates using predicates available on all platforms).

In fact, such bulk operations are widely applicable. For example, we have implemented joins for parallel data processing using a similar approach in the following chapters (also see [29, 31]). Namely, tuples of an input relation are redistributed to all the computation nodes. From that basis, local hash tables are created for lookup conducted by the other relation. In such application scenarios, the following three questions arising from the proposed framework are becoming to be interesting:

- performance: *will the responsible implementations be scalable and can they achieve comparable performance or even outperform the other two approaches?*
- parallelism: *how will the performance change with varying the number of threads over each hash table, if the whole available threads are fixed for a given system?*
- impact factors: *how will the high-level data distribution as well as the underlying hash table designs impact on the performance?*

The answers will give us an insight of the underlying hash implementation as well as an option to further improve the performance of applications using hash tables over distributed memory.

This chapter makes three main contributions. First, we propose a simple high-level parallel hashing framework, *structured parallel hashing*, targeting efficient processing of massive data on distributed memory. Second, we conduct a theoretical analysis of the scheme and present an efficient parallel hashing algorithm based on it. Finally, we evaluate on an experimental configuration consisting of up to 192 cores (16 nodes) and large datasets of up to 16 billion items (long integers). The experimental results demonstrate that the proposed approach is efficient and scalable. It is orders of magnitude faster than conventional distributed hashing methods, and also achieves comparable performance with a shared memory supercomputer-based approach, on a socket-for-socket basis.

The rest of this chapter is organized as follows: In Section 4.2, we conduct a theoretical analysis of different hashing frameworks. We present an efficient parallel hashing algorithm in Section 4.3. In Section 4.4, we experimentally evaluate our work, followed by a discussion in Section 4.5 and the conclusions in Section 4.6.

4.2 Theoretical Analysis of Hashing Frameworks

In this theoretical analysis, we make four assumptions: (1) our hash function produces a uniform distribution, (2) slot accesses after a hash collision follow a uniform random distribution, (3) each node can communicate with multiple remote nodes at the same time, and (4) the memory access and data transfer inside a physical node is zero (compared to the network-based operations). The first two assumptions are popular in currently theoretical studies [36] and the latter two are natural for an ideal distributed system. In addition to this, we refer to the distributed and thread-level hashing frameworks as HF1 and HF2 respectively, and our structured parallel hashing framework as HF3.

In general, the total time cost T to insert N items in a framework can be divided into three parts: distribution time for item transfers across memory resident in different nodes t_m, time for probing t_p and time costs due to memory contention t_c . As threads work in parallel in each framework, T would be the same as the time t by a single thread (assuming equal load). Specifically, we have $t_m = 0$ for HF2 as there is only a single shared memory location.

4.2.1 Distribution

We assume that the time cost of moving an item to the node itself is 0, and the time $t(x)$ to transfer x items to a remote node is $t(x) = \delta_0 + \delta_1 \cdot x$, where δ_0 is a constant that represents the latency for each data transfer[1] while δ_1 is the time for transferring a single item.

In a cluster with n physical nodes in which each has a constant number of threads e, there will be ne hash tables in HF1 and n in HF3, and each thread will process N/ne items. Since the items are processed one by one in HF1, the number of item transfers will be N/ne. In HF3, items are grouped into n chunks by each thread (namely total $ne \cdot n$ chunks with N/n^2e items each) and moved to the corresponding n nodes. Since the ratio of moved items to a remote node is $(n-1)/n$, the item transfer time in HF1 and HF3 is:

$$t_{m_1} = \frac{n-1}{n} \cdot \frac{N}{ne} \cdot (\delta_0 + \delta_1) \tag{4.1}$$

$$t_{m_3} = \frac{n-1}{n} \cdot n \cdot (\delta_0 + \delta_1 \cdot \frac{N}{n^2 e}) \tag{4.2}$$

This indicates that: (1) if n is a constant, t will be $O(N)$, and (2) for a given N, t will be

[1] Note that connections for data transfer could be retained, regardless, extra time cost for remote accesses still exist, such as memory allocation etc.

$O(n)$. Additionally, if n is fixed, the time difference $(t_{m_1} - t_{m_3})$ between HF1 and HF3 will be $O(N)$. It means that *with the increment of N, HF3 will spend less time on item transfers than HF1*.

4.2.2 Slot Probing

In each framework, threads insert items using a pseudo-random probe sequence. For a successful insertion, the last probed slot is empty, while the slot accessed before (if any) is occupied. For a hash table with c slots and v elements (load factor at end of execution $\alpha = v/c \leq 1$), according to the theorem for standard hashing presented in [36], we have the function between the average number of probes l in a successful search and v:

$$l(v) = \frac{1}{\alpha} \sum_{i=c-v+1}^{c} \frac{1}{i} \tag{4.3}$$

HF1 and HF3 implement insertion on individual partitions of distributed memory. Therefore, we have $v_1 = N/ne$ and $v_3 = N/n$. Moreover, for the single node with e' threads in HF2, there exists $v_2 = N$ and each thread processes N/e' items. Normally, we have $l(v_1) = l(v_2) = l(v_3) = l_0 \approx (-1/\alpha) \cdot \ln(1-\alpha)$, because N is a great number (for example 16 billions in our experiments) and there is $N >> ne$. If the time for a single probing operation is η_0, equal in each framework, then with the same load factor α, the probing time for a single thread would be:

$$t_{p_1} = t_{p_3} = \eta_0 l_0 \cdot \frac{N}{ne} \tag{4.4}$$

$$t_{p_2} = \eta_0 l_0 \cdot \frac{N}{e'} \tag{4.5}$$

This implies that for a given underlying platform, the probing time of each framework will be $O(N)$. And *for a fixed input, HF1 and HF3 can reduce the probing time by increasing the number of nodes n*.

4.2.3 Memory Contention

We define a *conflict* as the situation where more than one thread try to access the same hash table slot at the same time. The probability that a thread accesses a specified slot of a hash table (c slots and v elements) is $1/c$. With w threads, the probability that i ($1 \leq i \leq w$)

threads access the same slot would be:

$$p(v,i) = \binom{w}{i} \left(\frac{1}{c}\right)^{i} \left(1-\frac{1}{c}\right)^{w-i} \tag{4.6}$$

There will be $i-1$ thread conflicts when i threads access a same slot. Under the condition that $w << v$, the average number of conflicts for probe operations for a thread would be:

$$\begin{aligned} c(v,w) =& \sum_{i=1}^{w}(i-1)p(v,i) \\ =& p(v,2) + \sum_{i=3}^{w}(i-1)p(v,i) \\ \approx& p(v,2) \approx \frac{w(w-1)\alpha^2}{2v^2} \end{aligned} \tag{4.7}$$

For uniformly expressing the cost of the three approaches, here we refer to the number of items processed by each thread as $h_k v_k$, where the subscript k means the identify of each framework, namely $k = 1,2,3$. Then, we have $h_1 = 1$, $h_2 = 1/e'$ and $h_3 = 1/e$, which are all constant. If we assume that the waiting time λ_0 resulting from a single conflict in each framework is the same and there are $\bar{w_k}$ threads accessing a hash table, then, with the average number of probings described previously, we have:

$$t_{c_k} = \lambda_0 \cdot l_0 h_k v_k \cdot c(v_k, \bar{w_k}) = \frac{\lambda_0 l_0 \alpha^2 h_k}{2} \cdot \frac{\bar{w_k}(\bar{w_k}-1)}{v_k} \tag{4.8}$$

With a limited[2] n, e and e', $\lambda_0 l_0 \alpha^2 h_k \bar{w_k}(\bar{w_k}-1)/2$ will be a limited constant. Because v_k is $O(N)$, the time t_{c_k} will be $o(1/N)$. It means that *when processing very large-scale data, the time cost for memory contention can even be neglected in all frameworks*.

4.2.4 Performance Comparison

When processing large data ($t_{c_k} = 0$), the time difference $\Delta T_{ij} = \Delta T_i - \Delta T_j$ between HF3, HF2 and HF1 is:

$$\Delta T_{13} = \delta_0 \cdot (n-1) \cdot (\frac{N}{n^2 e} - 1) \tag{4.9}$$

$$\Delta T_{23} = \eta_0 l_0 \cdot (\frac{N}{e'} - \frac{N}{ne}) - (n-1) \cdot (\delta_0 + \delta_1 \cdot \frac{N}{n^2 e}) \tag{4.10}$$

[2]For example, for the cluster we use in our experiments, there is $n = 16$ and $e = 12$. For a supercomputer, the e' could be hundreds or thousands.

With a limited n, e and $e' \geq 2e$, if we set $N \to \infty$, then we have: (1) there is always $\Delta T_{13} > 0$ and (2) let $k_1 = \eta_0 l_0 / e$ and $k_2 = k_1^2 + (\delta_1/e)^2 + (2 - 4e/e') \cdot k_1 \delta_1 / e$, there will be $\Delta T_{23} > 0$ when

$$n \geq \frac{e'}{2k_1 e} \cdot (k_1 + \frac{\delta_1}{e} + \sqrt{k_2}) \tag{4.11}$$

This implies that *when processing a very large data set, (i) our hash framework is always faster than HF1, and (ii) it can perform better that HF3 with increasing the number of computation nodes*, at least based on a high-level theoretical analysis and on a simplified model. This assertion will be tested in the experimental evaluation.

4.3 Parallel Hashing

In this section, we present an efficient parallel hashing algorithm based on our framework. We focus on techniques to (1) maintain consistency in the distribution phase, and (2) avoid hash collisions and memory contention during hash operations. Additionally, motivated by the performance of data storage and information lookups in our applications (namely encoding, joins and indexing) [27, 29, 31, 33, 34], we just focus on the hash operations of insertion and searching.

4.3.1 Distribution

For an n-node system and t threads per node, all the threads read and distribute items in parallel. We introduce an integer parameter i to subdivide items based on a common $(n \times t)$-based hash partitioning, namely set $h(key) = key \ mod \ |n \times t \times i|$ to group and distribute items, based on the hash values of their *key*. Then, groups with hash values in the range $[k \cdot (t \cdot i), (k+1) \cdot (t \cdot i)]$, are sent to the k-th node ($k \in [0, n-1]$).

After the distribution, each computation node (rather than thread) has total $t \times n \times (t \times i)$ chunks of data to be processed locally. We treat all of them as a data **cuboid** where each chunk is indexed by (t, m, n), which represents that the chunk comes from the t-th thread with the hash value m at the n-th node. The detailed implementation at each node is given in Algorithm 1. The array *item_c* is used to collect the grouped items, and its size is initialized by the number of thread t and the modulo value m. Since each thread manages its own items, the reading and distribution operations can be performed in parallel across threads.

It is obvious that, for a given input dataset, the size of the cuboid (which is proportional to the number of received items for even data distributions) at each node will be constant, no matter how large the parameter i is. Nevertheless, with a larger i, the chunks of the cuboid

Algorithm 1 Item distribution at each node

```
1: Initialize items_c:array[array[array[item](m)]](t)
2: for i ∈ threads async do
3:     Read in item file f
4:     for item ∈ f do
5:         des ← hash(item.key)
6:         items_c(i)(des).add(item)
7:     end for
8:     for j ← 0..(m − 1) do
9:         Push items_c(i)(j) to r_items_c(i, j, k) at node k, where k = m/n
10:    end for
11: end for
```

would be more fine-grained. This means that the size of the transferred data as well as the allocated memory each time will be smaller. Furthermore, if we process the cuboid data at the unit of a chunk, the workload of each local thread would also be more uniform. However, inter-node communication will become more frequent, negatively affecting performance. We will examine this trade-off in our evaluation.

4.3.2 Processing

Since the number of received items at each node can be easily recorded in the distribution phase, we can directly allocate the required size hash table and initialize it. In the meantime, during key insertion, there exist various hashing strategies to minimize hash collisions and different mechanisms like the lock-based and non-blocking approaches are proposed to address the problem of memory contention [36, 58, 59]. As we focus on the parallelism over hash tables, we adopt *linear probing* for hash collisions and *CAS* (compare-and-swap) for memory contention in our implementations, which is very popular in recently studies [41, 50, 51, 59, 106, 131]. In addition, we propose a new *range-based* algorithm, aimed at removing memory contention.

CAS

Compare-and-swap ensures the slot for the key that it is about to insert does not have another key inserted during its operation [106]. As shown in Figure 4.4, the hash table is initialized by two arrays. One array is used to hold the items, and other one is a status array using CAS to indicate whether the corresponding slot in the former array is filled or not. In the operation of insertion, the slot of an item is located by its hash value. The thread first checks

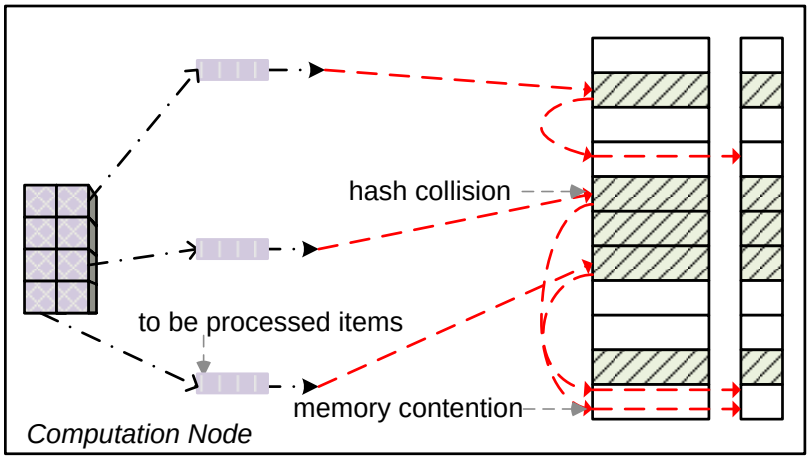


Fig. 4.4 CAS-based Implementation.

whether that slot in the item array is filled or not. If not, the thread would atomically check and set the slot at the same index of the status array. If the slot is already set, the thread will continue to the next slot.

The details of our implementation is shown in Algorithm 2. We use the array *ht* to store the items while the array *stat* is used to indicate occupancy. Each slot in the stat array is initialized with the element `atomicBoolean` to support the CAS operations. After that, the received data chunks will be scheduled as a task queue and assigned to all the available threads. For each item, the initial location of the slot will be calculated by a hash function $h_1(k)$. The empty slot searching process will start with the position $h_1(k)$ of *ht*. If a slot is not occupied and the CAS operation over *stat* also returns the value of *true*, then the item will be inserted, otherwise, the next slot will be probed. We use a modular arithmetic to cycle the location of an array from the bottom to the top and the searching process will be repeated until a free slot is found. The insertion progress will be ended when all the inserting tasks are finished.

There are two possible issues when using CAS: (1) the *ABA* problem as described in [59]. The key value of a slot in our implementation only changes from *null* to another value and never changes back again, as the same as the scenarios in [106], therefore the ABA problem could not exist in our implementations; and (2) the *contention hot spots* problem as presented in [41]. In fact, this problem becomes a performance issue because [41] focuses on the study of continuously changing the same variable with multi-threads. In contrast, threads in our method do not work on a specified slot but over the whole table instead. From the probability as we analyzed in Section 4.2, it is clear that the performance of our implementations will be not affected by such an issue, at least for the massive uniform distributed data.

Algorithm 2 CAS-based implementation at each node

```
1:  Allocate buffers for hash table ht:array[item],
      stat:array[atomicBoolean](true),
2:  for each r_items_c(i, j, k) async do
3:      for item ∈ r_items_c(i, j, k) do
4:          e ← h1(item.key)
5:          Search an empty slot start from the eth position
6:          if ht(e).null ∧ stat(e).CAS(true,false) then
7:              ht(e) ← item
8:          else
9:              e++, Continue searching
10:         end if
11:     end for
12: end for
```

Range

We propose the *range-based* approach from the basis of the *parallel radix join* [20] algorithm, which is commonly used in recent research targetting efficient parallel joins [16, 20]. The main idea is that the subdivided data is assigned to individual threads and then each thread processes the data independently. Regardless, the method for *joins* focuses on workload assignment at the hardware-level, such that the size of data chunks is set to the cache size so as to minimize the *cache miss* etc. Compared to that, our approach is concentrated such that all the threads can work on a given hash table without any influence from each other.

In general, as demonstrated in Figure 4.5, we map chunks of data in the cuboid to the

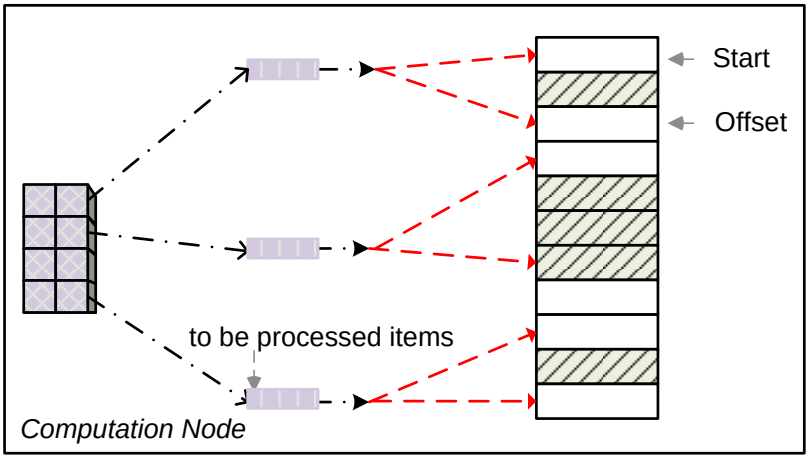


Fig. 4.5 Range-based Implementation.

specified hash table according to the value of index m as we described previously. Because we can easily calculate the size of the mapped chunks, the mapped *range* on a hash table can be simply presented by two values: the start point *start* and the size *offset*. If threads at each node process the items in the unit of chunk section (according to index m), then all of them would work in a specified range, and no memory contention happens.

The detailed implementation at each node is presented in Algorithm 3. We first compute the size of each range by S_k=$\sum_{i,j} item\_c(i,j,k).size$. When inserting an item, three parameters - the item, the start slot R_{k-1} and the end slot R_k of the range, are transferred to the hash function h_2 to locate the start probing point in the hash table. Similarly, we also use a modular arithmetic to ensure that the probings work in the specified range. The program is terminated when all the places finish item insertion.

Algorithm 3 Range-based implementation at each node

```
1: Compute the k th Range:S_r
2: The end slot of the k th Range R_k=∑_{r<k} S_r
3: Allocate buffers for hash table: ht:array[item]
4: for each items_c(i,j,k) async do
5:     for item ∈ items_c(i,j,k) do
6:         e ← h_2(item.key, R_{k-1}, R_k)
7:         Searching a empty slot start from the e th slot
8:         if ht(e).null then
9:             ht(e) ← item
10:        else
11:            e++, Continue searching in [R_{k-1}, R_k]
12:        end if
13:    end for
14: end for
```

The calculation of each *range* depends on the distribution of values in the hashtables, which can be easily computed in our data distribution phase. Therefore, the proposed *range* method fits our framework well. We will evaluate its performance and compare it with the popular CAS method.

Searching

Searching is very similar to the insertion operation as described. Keys are mapped to locations in the same way as in Section 4.3.1 and threads can independently search on the hash table. Because no thread synchronization is required in this phase (assuming no concurrent writes), threads in the CAS-based implementations can freely access any slot without

checking the status array. For the range-based implementation, an additional operation is required to read the range.

4.4 Evaluation

Platform. Our evaluation platform is the *High-Performance Systems Research Cluster* located at IBM Research Ireland. Each computation unit of this cluster is an iDataPlex node with two 6-core Intel Xeon X5679 processors running at 2.93 GHz, resulting in a total of 12 cores per physical node. Each node has 128GB of RAM and a single 1TB SATA hard-drive and nodes are connected by Gigabit Ethernet. We implement our algorithms with the parallel language X10 [22] over the RHEL with Linux kernel 2.6.32-220. We use X10 version 2.3 and compiling it to C++ over gcc version 4.4.6.

Dataset and Metric. Table 4.1 shows the input and output parameters for our experiments, with bold font indicating default values. We have generated several datasets up to 16 billion integers. Data follows a uniform distribution when *Zipf factor* is equal to 0, or a skewed distribution with the associated α parameter. We mainly measure the runtime of each test in terms of: *distribution time*, *insertion time*, *hashing time* and *search time* as described. In the meantime, two types of hash tables based on our framework are examined: (1) *Structured Distributed Hash Tables (`SDHT`)*, in which there is a single thread per logical computation node. Therefore, this kind of hash table does not suffer from memory contention, but at the cost of reduced flexibility in terms of load balancing. (2) *Hybrid Parallel Hash Tables (`HPHT`)* have multiple threads per logical node operating with the CAS or *range* strategies (referred to as Range in the following) as described before.

In the following, we first conduct the performance comparison of each hash framework on a basic test. Then, we evaluate the scalability of `SDHT` and compare the performance of `HPHT` using different lock-free strategies. Finally, we study the impact factors of our hash tables and compare our results with current implementations as presented in [85] and [51]. Because the standard deviation between executions was very small in our tests, we record the mean value based on ten measurements.

4.4.1 Comparison of Frameworks

We conduct a simple performance comparison of the three hashing frameworks already described based on the CAS strategy. We implement the thread-level parallel one a single machine with 12 node, and other two parallelism on a distribute system with the same node,

Table 4.1 Experimental Parameters.

Input Parameters	
Parameter	**Values**
Hash table implementation	`SDHT`, `HPHT`, CO_U, RHH_U
Dataset size (billions)	0.5, **1**, 2, 4, 8, 16
Zipf factor	**0**, 0.2, 1, 1.8
Load factor	0.6, **0.75**, 0.9
#Threads	12, 24, 48, 72, 96, 120, 144, 168, **192**
#Threads/Table	**4**, 12
#Threads/Core	**1**
Hash Collision Strategy	CAS, **Range**
i parameter	**1**, 10, 100
Key length	32 bits, **64 bits**

Output Parameters	
Parameter	**Description**
Read time	Time to read data from disk
Distribution time	Time to distribute items
Insertion time	Time to insert to hash-table
Hashing time	Sum of Distribution and Insertion time
Search time	Time to search for all items

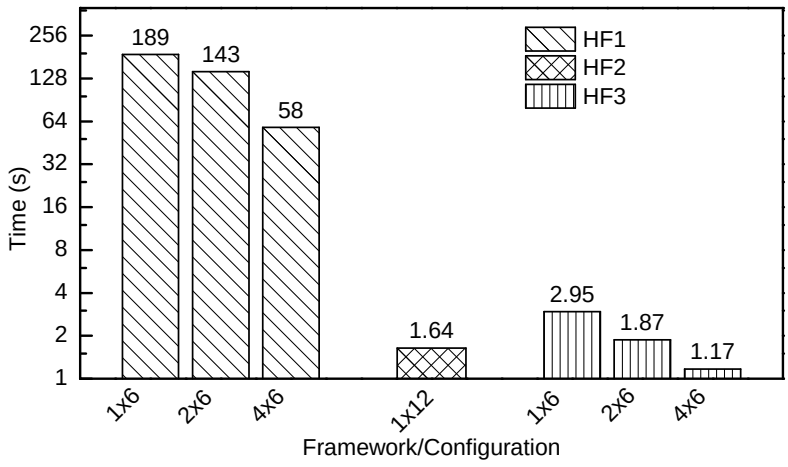


Fig. 4.6 Performance comparison of three frameworks.

but just use 6 cores each machine (namely a small machine). We process 10 million keys and present the result in Figure 4.6. There, the configure 2×6 indicates a configuration of two machines using 6 cores each. It can be seen that our framework *HF3* performs much better than *HF1*. However we are slower than *HF2* initially but when using 4 machines (24 cores), our implementation become faster. All this is consistent with our theoretical analysis in Section 4.2.

4.4.2 Structured Distributed Hash Tables

We test the scalability of our *SDHT* by varying the number of processing threads and the size of input data. The results are shown in Figure 4.7. We can see that the time cost for inserting and searching is almost the same and linear with the number of nodes. For a small number of threads, distribution time is not linear, since for a single node there is no network communication and for two nodes (24 threads), only 50% of the data needs to be transferred over the network. With more than 72 threads, the distribution cost decreases linearly with the number of nodes. Overall, the hashing time follows the same pattern.

To study the scalability of our algorithm with increasing input size, we fix the number of threads to 192 (16 nodes), start our tests with 500 million integers and repeatedly double the size of the input until 16 billion. The results are presented in Figure 4.8. The hashing time is linear with the size of the input, and nearly matches the ideal speedup scenario. The same holds for distribution, insertion and searching. Furthermore, the time spent in the insertion phase is nearly the same as the searching phase, and both are less than that of the distribution phase.

From the results above, we can see that hash table construction scales very well both with the number of threads and the input size. We also notice that, with a 16-node cluster, the item distribution costs about 60% more than insertion and searching. We will characterize the possible factors in Section 4.4.4.

4.4.3 Hybrid Parallel Hash Tables

We elaborate on the performance of `HPHT` for different strategies and input parameters. `HPHT` uses multiple threads per node (could be logical or physical), and by extension, multiple hashtables. We choose two typical cases: four threads and three *logical nodes* per physical node (4×3) and twelve threads and one *logical node* per physical node (12×1). We further experiment with regard to scalability with the number of threads.

Figure 4.9 presents runtimes to process 1 billion integers. Similar to `SDHT`, both the CAS

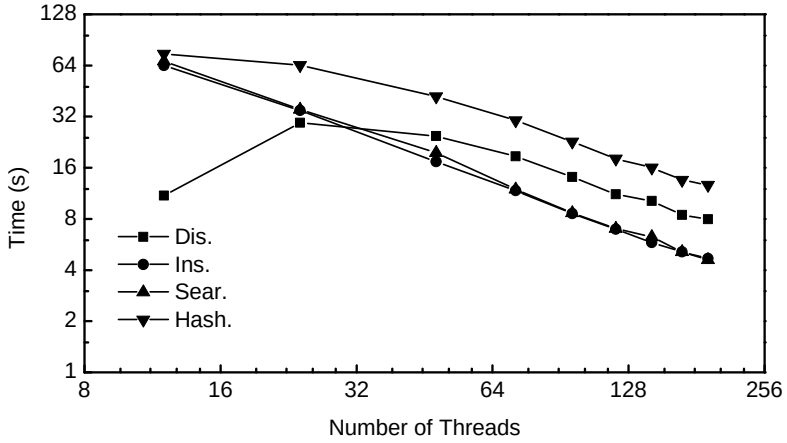


Fig. 4.7 Time cost with varying number of threads for SDHT.

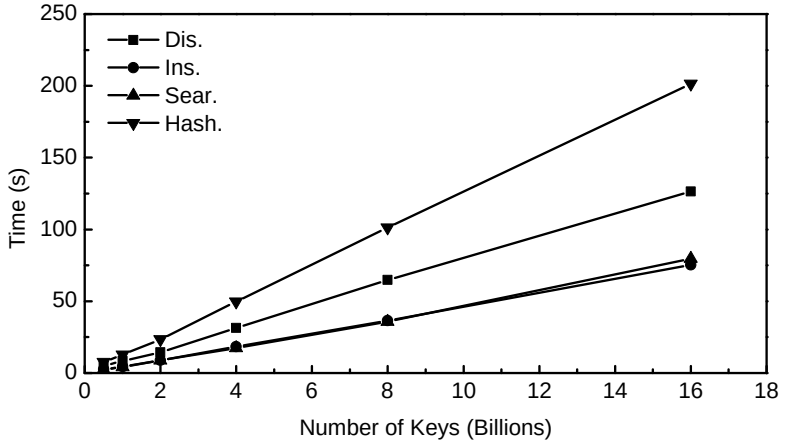


Fig. 4.8 Time cost with varying size of input for SDHT.

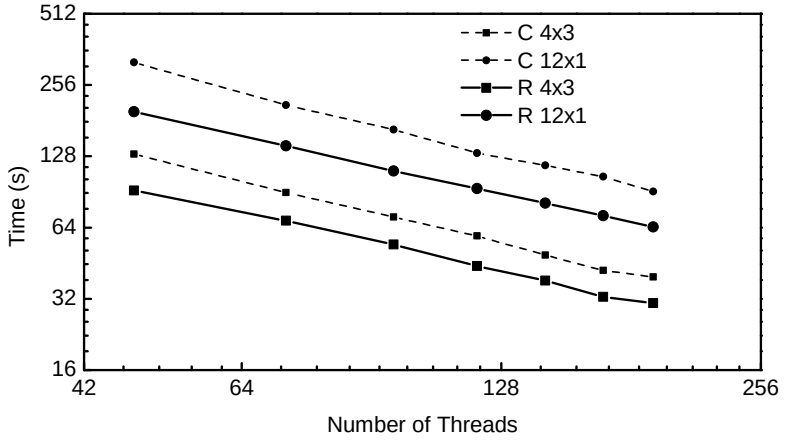


Fig. 4.9 Time cost with varying the number of threads for HPHT.

and Range implementations scale well with the number of threads. With detailed runtime comparison, we find that the proposed *Range* method performs much faster than CAS, both for insertion and searching. There are three possible reasons: (1) hash table construction in CAS is more complex (using extra-arrays); (2) there is extra atomic compare-and-swap operations in CAS while there is no memory contention in Range; and (3) regarding search, although there are no compare-and-swap operations for CAS, Range still benefits from superior memory locality for individual threads.

Given a fixed number of threads, the implementation configured with 12×1 performs worse than 4×3 for each algorithm, and both of them are slower than `SDHT`. Although `HPHT` is slower than `SDHT` when processing uniformly distributed integers, `HPHT` scales equally well. Moreover, `HPHT` features thread coordination, which would be advantageous in some scenarios, such as against the data skew.

To validate this claim, we conduct a test on 16 nodes under dataset skew. Each dataset contains 1 billion integers following the Zipf distribution ($\alpha = 0.2$, 1 and 1.8). To support the thread coordination in Range operations, we also set the parameter i to 10 in each implementation (recall that threads in Range operations process data chunks according to the value of modulo, so there will be no thread coordination if $i = 1$). As shown in Figure 4.10, distribution time and insertion time increases with the skew of the dataset for all settings. However, for $a \geq 1$, `HPHT` significantly outperforms `SDHT`, indicating superior load balancing, mainly during insertion. Additionally, the configuration with 12×1 is still slower than 4×3, which means that hash tables with moderate parallelism could be a better choice even in the presence of high skewed data.

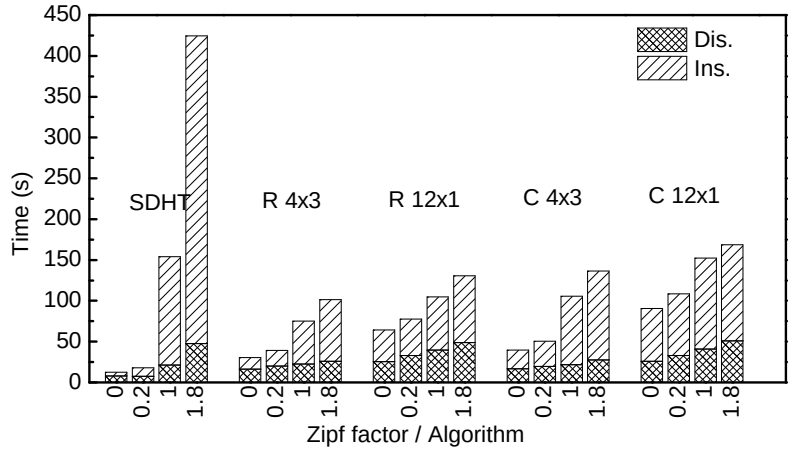


Fig. 4.10 Runtime by varying *Zipfian factor* in each implementation.

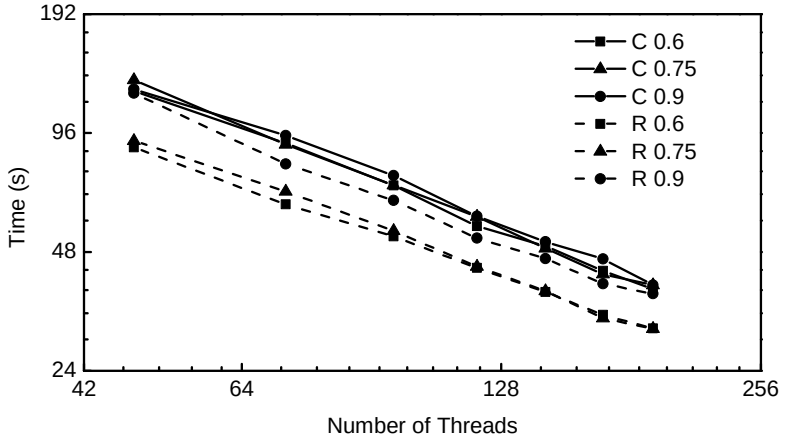


Fig. 4.11 Time cost with different load factors.

Table 4.2 Detailed Time cost of processing different integer lengths

# Threads	64-bit integer (sec.)			32-bit integer (sec.)		
	Dis.	Ins.	Sear.	Dis.	Ins.	Sear.
48	38.63	53.05	55.65	18.26	55.39	56.91
72	29.02	39.28	38.26	15.03	37.44	36.24
96	25.51	28.80	28.77	13.22	27.66	28.94
120	21.23	22.84	22.78	11.34	22.27	24.01
144	19.78	18.46	20.10	10.47	18.61	18.92
168	16.37	16.26	16.74	9.33	16.00	17.01
192	16.40	14.26	14.19	9.98	14.12	15.01

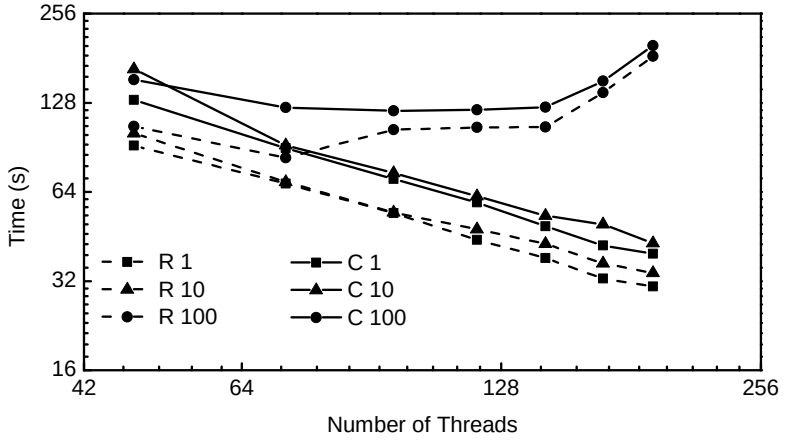


Fig. 4.12 Time cost with varying the parameter *i*.

4.4.4 Impact Factors

We also test factors with potential performance impact. As the curves with different numbers of threads per place are nearly the same, both in `SDHT` and `HPHT`, we only present results with the configuration 4×3 based on CAS and Range implementations. The factors we have considered are (a) the *load factors* of hash tables, (b) the *length of the processed keys* and (c) the parameter i, mentioned in Section 4.3.1.

Three different values for load factor (0.6, 0.75 and 0.9) are examined in our tests. Figure 4.11 shows the hashing time for 1 billion integers. Once again, we observe that all the implementations scale well with the number of threads. Moreover, as expected, hashing time increases with the load factor. For both strategies, the runtime with load factor 0.6 and 0.75 is nearly the same. For a load factor of 0.9, in Range, runtime increases by nearly 20% while for CAS, it increases only by 3%. This also indicates that hash collisions have a more significant effect on the performance for the Range implementation. There is a trade-off between the memory consumption and the load factor, therefore, in real implementations, assigning the load factor to 0.75 would be a better choice for Range and 0.9 if using CAS.

We test the time of processing 1 billion integers represented with 32 bits or 64 bits. Because the CAS and Range implementations show the same characteristics, we only present the execution time for the Range algorithm as shown in Table 4.2. The time spent on distributing the 32-bit integers is about a half of that for the 64-bit objects, while the insertion and the search time do not change. This is in contrast with the conclusion in [85] that varying the size of integers has no effect on time. This difference shows the essential difference between our implementation and other general algorithms: we used a high-level structured method to group items that need to be sent to the remote nodes, while other methods send many short messages that overwhelm the network, leading to significant inter-node communication and coordination overhead. This can also be observed in our results in that the distribution with 168 threads and 192 threads takes nearly the same time, because the transferred data chunks become too small.

Finally, we evaluate how the data partitioning in the distribution phase affects the execution time. The parameter i is set to 1, 10 and 100 respectively and the results are present in Figure 4.12. The runtime in both CAS and Range with $i = 10$ is slightly greater than that with $i = 1$, and is fairly linear with the number of threads. However, when setting the parameter to 100, the time cost decreases at first and then increases with the number of threads, leading to bad scalability. The decrease in the size of transferred data for each thread at the beginning reduces the distribution time, but as the number of threads increases, the vastly increased number of chunks incurs significant coordination overhead. The above

result, together with our experiments regarding skew, indicate that higher *i* should be chosen for larger data sizes and higher skew.

4.4.5 Comparison with Current Implementations

The latest evolution with *distributed parallel hashing* is reported in [85], using 768 threads on a cluster to process 19.2 million items takes 18.2 secs using UPC and 27.4 secs using MPI. In comparison, our best performing implementation can process 1 billion items in just 13 secs with 192 threads, and we also achieve linear scale with the increment of threads. This is similar as the results presented in Figure 4.6, and the great difference evident arises from the different hashing frameworks utilised.

Table 4.3 Comparison with results presented in [51] (time in seconds)

Algorithm	**16 Sockets**		**32 Sockets**	
	Read.	**Hash.**	**Read.**	**Hash.**
Cray CO_U	123	90	123	46
Cray RHH_U	124	150	123	77
SDHT	57	113	30	59
Range 4×3	70	257	32	152
Range 12×1	72	570	36	301
CAS 4×3	68	331	32	192
CAS 12×1	72	842	37	466

We also conduct a detailed comparison with the fastest performing implementation in the literature, presented in [51]. [51] implements *thread-level parallel hashing* on a Cray XMT supercomputer using two techniques: CO and RHH[3]. Although the approach in [51] also optimizes hashing of skewed loads, it is not the focus of this thesis. Table 4.3 shows the file reading and hashing time to process **5 billion** integers. The Cray XMT is a shared-memory architecture using a specialized interconnect and a latency-tolerant model. Since a direct comparison of processor speeds in not meaningful, we group the results on a per-socket basis. We observe that, SDHT is faster than RHH and slower than CO if we do not consider the reading time. If we consider reading time, SDHT is faster than all other techniques and systems. HPHT is slower than all other approaches, but still remains within an order of magnitude of the best performing system. Overall, although our system relies exclusively on low-cost commodity hardware, we observe that it achieves comparable performance to

[3]the results presented here were obtained by communication with the authors

a shared-memory system using a specialized interconnect and processor architecture. With increasing nodes, it is expected that we can even outperform [51] on the *hash* operation on the basis of the theoretical analysis in Section 4.2.

4.5 Discussion

The study of distributed parallel hashing main focuses on (1) low-level communication schemes such as the use of the IBM LAPI [83], and (2) parallel programming paradigms or languages, such as the use of Java, MPI and UPC [43, 85, 98]. In these implementations, hash operations are always accompanied with frequent and irregular remote memory access with a concomitant increase in low-level communication overhead and the associated performance hit. Therefore, they are more suitable for processing small data, but not for massive data.

There is long history of theoretical studies [72, 84] in terms of the thread-level parallel approaches. By employing different hashing strategies, implementations on various platforms have achieved excellent performance [50, 51, 106]. Our implementation performs comparably or slightly worse than the fastest one [51], however our approach relies on low cost commodity hardware, adding to its flexibility.

GPU computing has become a well-accepted parallel computing paradigm and there are many reports on implementations of parallel hashing based on that [8, 48]. Implementations of these hash tables exhibit strong performance. However, GPU memory is limited so therefore such methods cannot work with excessively hash tables of the sizes shown in this thesis. In addition, reading data into GPUs takes a considerable time, adding significant overhead for a simple task, from the perspective of computation.

Although *parallel hash joins* are widely studied in modern parallel database management systems [7, 16, 20], there is little research focuses on the parallelism of underlying hash tables. With the increase in size of process datasets in this domain [7], we expect that the hash strategies used in our hash tables can further improve join performance here.

The idea behind our method is straightforward, yet not trivial, and does not appear in the literature. Consequently we believe that the evaluations conducted here and the results described are of value to the community as a basis for understanding the merits of the approach. Moreover, our theoretical analysis in Section 4.2 confirm that our structured method is faster for large datasets - a result verified through our experiments. Finally we also contribute a *range-based* strategy for our hashing implementation, which is shown to be faster than the commonly used *CAS* method within our framework.

4.6 Conclusions

In this chapter, we proposed a high-level structured framework for parallel hashing, which has been designed for processing massive data. This framework supports (a) distributed memory while avoiding frequent remote memory access, and (b) thread coordination on a per-partition basis. Based on that, we presented an efficient parallel hashing algorithm by employing the popular CAS and our proposed *range-based* lock-free hashing strategies.

The experimental evaluation results show that our implementation is highly efficient and scalable in processing large datasets. Moreover, this hash framework demonstrates useful flexibility in that it can employ various hashing techniques and can be run on commodity hardware. Finally, the proposed Range lock-free strategy is faster than the conventional CAS operation and presents better load balancing characteristics than approaches which use a single thread per partition. Additionally, we have characterized the performance of our hash implementations through extensive experiments, thereby allowing us to make a more informed choice for our high-performance implementations over distributed memory.

In the following chapters, we will present a detailed implementation of our proposed framework, and focus on techniques to improve the performance for the three core operations (encoding, joins and indexing) we have described.

Chapter 5

Scalable RDF Data Compression using X10

5.1 Introduction

With the study of triple stores and parallel hashing in the previous two chapters, now we turn to the detailed implementations of our framework. In this chapter, we will propose an efficient dictionary encoding method to compress large RDF data in parallel. Our solution is based on a distributed architecture with multiple dictionaries. Namely, the RDF data is partitioned and then compressed using a dictionary on each computation node. However, similar to the state-of-art MapReduce method [116] as described in Chapter 2, there exist three main challenges under this schema:

- Consistency - a term appearing on different compute nodes should have the same id.
- Performance - ensuring consistency based on naive methods can lead to serious performance degradation.
- Load balancing - the heavy skew of terms which characterizes real world linked data [78] may lead to hotspots for the nodes responsible for encoding these popular terms.

Both in space and time, the mapping of a term need always keep its uniqueness. For example, once the term *"dbpedia:IBM"* is first encoded as id *"101"* on node A, when encoding this string on another node B, we should also use the same value *"101"*. Hash functions are potentially useful, but the length of the hash required to avoid collisions when processing billions of terms makes the space cost prohibitive.

We can ensure the consistency of the compression in the above example by copying the mapping *[dbpedia:IBM, 101]* from node A to node B, but network communication cost and dealing with concurrency (e.g. locking on data structures) would lead to low performance.

Compared with the two issues above, load balancing presents a bigger challenge as the distribution of terms in the Semantic Web is highly skewed: there exist both popular (like predefined RDF and RDFS vocabulary) and unpopular terms (like identifiers for entities that only appear for a limited number of times). For a distributed system, like ours, any compression algorithm needs to be carefully engineered so that good network communication and computational load-balance are achieved. If terms are assigned using a simple hash distribution algorithm, the continuous re-distribution of all the terms would undoubtedly lead to an overloaded network. Furthermore, popular terms would lead to load-balancing issues.

For the sake of explanation, let us categorize terms into three groups: high-popularity terms that appear in a significant portion of the input triples, low-popularity terms that appear less than a handful of times and average-popularity terms (which is also the largest portion of RDF data). The state-of-the-art MapReduce compression algorithm [116] efficiently processes high-popularity terms. The very first job in the algorithm is to sample and assign identifiers to popular terms, using an arbitrarily chosen threshold. These identifiers are then distributed to all nodes in the system, and used to encode terms locally at each node. This dramatically improves load balancing and speeds up computation. For the rest of the terms, the data is repartitioned, and identifiers are assigned. For low-popularity terms, this also works well, as there are not many redundant data transfers. For low-popularity terms, we can either retrieve their mappings (possibly for multiple nodes), or we can send the data to the node where it is going to be encoded. In either case, the number of messages will be limited. For medium-popularity terms, the situation is different: Assume a term that appears 10000 times, and we have 100 compute nodes. If all nodes would need to retrieve the mapping from a single node, we would need 200 messages. If we repartition the terms, we would need at least 10000 messages. One can easily see the situation reversed for a term that appears 100 times (i.e. partitioning data might be more efficient that retrieving mappings). Then the question is: *how can we reconcile efficient encoding of popular and non-popular terms?*

To solve the above problems, we propose a straightforward but very efficient and scalable solution for compressing massive RDF data in parallel in the following. We develop an algorithm and present its detailed implementation using the X10 language [22]. We evaluate performance with up to 384 cores and with datasets comprising of up to 11 billion triples (1.9 TB). Compared to the state-of-the-art [116], our approach is faster (by a fac-

tor of 2.6 to 7.4), can deal with incremental updates in an efficient manner (outperforming the state-of-the-art by several orders of magnitude) and supports both disk and in-memory processing.

The rest of this chapter is organized as follows: Section 5.2 introduces the proposed RDF compression algorithm and the detailed implementation. Section 5.3 discusses optimizations and improvements for the algorithm. Section 5.4 describes the experimental framework and provides a quantitative evaluation of the algorithm. Section 5.6 concludes the work done in this chapter.

5.2 RDF Compression

In this section, we first describe the details of our RDF compression algorithm and its detailed implementation using X10 language. Then, we present its difference as well as its advantages compared to the state-of-art method.

5.2.1 Main Algorithm

We describe the implementation of an RDF compression algorithm on a distributed memory system. We use distributed dictionaries, one per place (recall that a place is a logical abstraction for an underlying processing element), for encoding the input data sets. Each data set is first divided into a number of *chunks* and assigned for processing on separate places. The initial partitioning of chunks is random. The overall implementation strategy for each place and the corresponding data flow are shown in Figure 5.1.

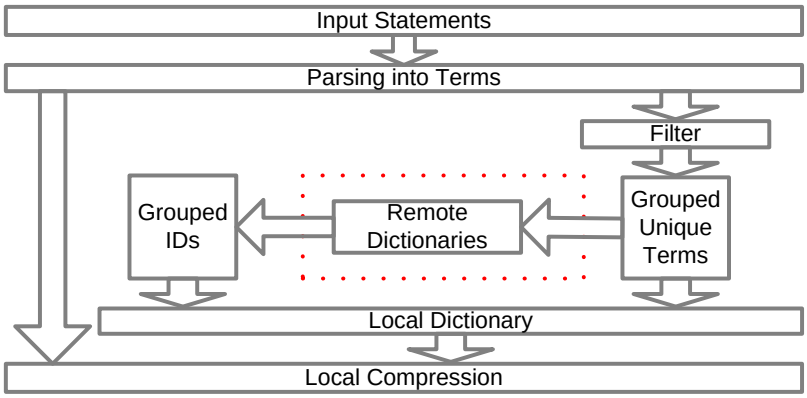


Fig. 5.1 Data flow of the RDF compression in our implementation.

First and foremost, every statement in the input set is parsed and split into individual

terms, essentially, the *subject*, the *predicate*, and the *object*. All these parsed terms are filtered to remove the replications, and the extracted *unique* terms are then divided into individual groups according to their hash values. The number of groups is set to the same as the number of places, and all terms in a aforementioned group have the same hash. In order to maintain consistency, the hash value of each term maps this term to a *single* dictionary in the distributed memory system where it gets encoded. The groups of unique terms are pushed to the dictionaries responsible for encoding these terms. Every place builds a local dictionary, for encoding, based on the grouped unique terms and the corresponding group of ids received from remote nodes. Once all terms are encoded the grouped *ids* are retrieved and the statements in the input data set are compressed.

5.2.2 Detailed Implementation

We divide the whole process into four phases and present their detailed implementations based on the X10 language as the following four steps.

Step 1: Initialization. We use the `DistArray` objects provided to implement our distributed data structures. The initialization for these objects, at each place, is shown in Algorithm 4.

In this process, the detailed meanings of the initialized objects at each place are:

- *dict* is the dictionary that maintains the term-id mappings during the whole compression process.
- *term_c* collects the terms and keeps them in sequence for subsequent encoding.
- *local_key_c* is the array that collects the groups of unique terms that need to be sent to remote places for encoding.
- *local_value_c* is the array that collects all the encoded unique ids from remote places. The sequence of ids in *local_value_c* is the same as terms in *local_key_c*, thereby making it easy to insert the terms and their respective encodings into the local dictionary.
- *remote_key_c* is a temporary data structure used to receive the serialized the grouped unique terms that are sent from remote places.

Step 2: Term Grouping and Pushing. After the parsing, we employ a `hashset` structure to process the parsed terms and to extract the unique ones that need to be transferred to

Algorithm 4 Dictionary encoding part I: Initialization

the number of places: P
Global initialize DistArray objects: *dict term_c*, *local_key_c*, *local_value_c*, *remote_key_c*

```
1: finish async at p ∈ P {
2: dict(here):hashmap[string,long]
3: term_c(here):array[string]
4: local_key_c(here):array[array[string]]
5: local_value_c(here):array[remote_array[long]]
6: remote_key_c(here):array[remote_array[char]]
7: }
```

Algorithm 5 Dictionary encoding part II: Filter and Push Terms

```
1: finish async at p ∈ P {
2: Initialize key_f:array[hashset[string]](P)
3: Read in file f_i
4: for triple ∈ f_i do
5:     terms(3)=parsing(triple)
6:     for j ← 0..2 do
7:         des=hash(terms(j));
8:         if terms(j) ∉ key_f(des) then
9:             key_f(des).add(term(j))
10:        end if
11:        term_c(here).add(term(j))
12:    end for
13: end for
14: Copy the terms in key_c(i) to local_key_c(here)(i)
15: for n ← 0..(P−1) do
16:    Serialize local_key_c(here)(n) to ser_key(n)
17:    Push ser_key(n) to remote_key_c(n)(here) at the place n
18: end for
19: }
```

remote places. This is done for all terms irrespective of their popularity. Using the `hashset` guarantees that any given term can possibly move to a remote place just once, per current place.

The detailed implementation is given in Algorithm 5. A `hashset` is initialized at each place. Each `hashset` collects terms according to their hash values. Before adding the parsed term into the *term_c* queue, a term is added to the `hashset`: *key_f*, if not already present. After processing all the triples, the filtered terms will be copied into *local_key_c*, and then serialized and pushed to the assigned place for further processing.

The structure *local_key_c* is kept in memory for the later local dictionary construction as shown in Figure 5.1. The serialization/deserialization process is used only when the push array objects are neither `long`, `int` nor `char`, otherwise we directly transfer the data. Since the terms collected by each `hashset` are the unique ones to be sent to remote places, the network communication and later computational costs are significantly reduced. We use the `finish` operation in this part to guarantee the completion of the data transfer at each place before the term encoding.

Step 3: Term Encoding. Once the grouped unique terms have been transferred to the appropriate remote places, the term encoding can commence. The term encoding implementation at each place is similar to sequential encoding. The received serialized char arrays, representing the grouped unique terms, are deserialized to string arrays. Then the terms in such arrays access the local dictionary sequentially to get their numerical ids. In this process, if the mapping of a term already exists, its id is retrieved, else, a new id is created, and the new mapping is added into the local dictionary. In both cases, the id of the encoded term is added into a temporary array for so that it can be sent back to the requester(s). The value of a new id is determined by the summation of the largest id in the dictionary and the value *P*, the number of places. This guarantees there is no clash between term ids assigned at different places. Furthermore, each id is formatted as an `unsigned` 64-bit integer in order to remove limitations regarding maximum dictionary size[1].

We also write out the new mappings in this phase, as they build up the final dictionary. Once the encoding of the grouped unique terms is complete, we shift the `activity` to the corresponding place where the terms originated, and retrieve the ids. We then proceed in processing the following group. All encoding happens in parallel at each place, and we use the `finish` operation synchronization. The details of the algorithm are given in Algorithm 6.

Step 4: Statement Compression The statements at each place can be compressed after all the ids of the pushed terms have been pulled back. Since the terms and their respective ids are held in order inside arrays, we can easily insert these mappings into the local dictionary. Once inserted, we encode the parsed triples in array *term_c*. Finally, we write out the ids to disk sequentially as shown in Algorithm 7. The whole compression process terminates when all individual activities terminate. Note that, in the actual implementation, we build a temporary hashmap to hold all the mappings and discard it after the encoding to optimize memory use.

[1] it is possible to use arbitrary- or variable-length ids in order to further optimize space utilization, but this is beyond the scope of our work.

Algorithm 6 Dictionary encoding part III: Encode Terms and Pull Back IDs

```
1: finish async at p ∈ P {
2: Initialize key_c:array[string], value_c:array[long]
3: for i ← 0..(P − 1) do
4:     Deserialize remote_key_c(here)(i) to key_c
5:     for key ∈ key_c(i) do
6:         if key ∈ dict(here) then
7:             value_c.add(id)
8:         else
9:             id = (dict(here).size + 1) ∗ P
10:            dict(here.id).put(key,id)
11:            value_c.add(id)
12:            Out-writing <key,id>
13:        end if
14:    end for
15:    at place(i)
16:    Pull value_c(i) to local_value_c(here)(i)
17: end for
18: }
```

Algorithm 7 Dictionary encoding part IV: Statement Compression

```
1: finish async at p ∈ P {
2: for i ← 0..(P − 1) do
3:     Add <key,id> from local_key_c(here)(i) and local_value_c(here)(i) to dict(here)
4: end for
5: for term ∈ term_c(here) do
6:     id = dict(here).get(term).value()
7:     Out-writing id
8: end for
9: }
```

5.3 Improvements

In this section, we present a set of extensions to our basic algorithm which improve efficiency and extend the applicability of the approach to a larger set of problems and computation platforms. The section concludes with a brief account of the theoretical complexity of our algorithm.

5.3.1 I/O and Data Transfers

X10 does not yet provide efficient I/O operation libraries for reading large data sets, as noted by Zhang et al. [130]. Moreover, using the standard `at{p}` construct for copying data incurs a substantial penalty for deep copying data structures. In order to alleviate these bottlenecks, Zhang et al. [130] recommend the use of `mmap` system call and `array.asycCopy` method. We adopt the latter approach and extend the first one with the `zlib` compression library to provide more efficient reading of large data sets.

Our preliminary experiments suggest that using *just* the `mmap` approach for large I/O operations scales well to medium sized data sets with less than hundreds of gigabytes of data. However, for very large data sets measured in tera-bytes, reading `gzip`-compressed files in memory and decompressing them on the fly results in substantially improved I/O performance. Moreover, compressing data in the `gzip` format also reduces disk space usage.

The X10 standard library does not provide any interface for reading and writing compressed gzip files, so we build a small library based on `zlib` and integrate it with our X10 code via the foreign function interface. We use the compressed datasets only while reading, since the resultant output is comparatively small and we simply write it out in bytes using the `OutputStreamWriter` class in the X10 standard library.

5.3.2 Flexible Memory Footprint

In our algorithm, the `DistArray` objects (Figure 5.1) are kept in memory throughout the compression process. This limits the applicability of the method to clusters with sufficient memory to hold all data structures in memory.

To alleviate this problem, we divide the input data set into multiple *chunks*, usually a multiple of the number of places. The corresponding code change is shown in Algorithm 8. The encoding process is divided into multiple loop iterations corresponding to each chunk. In each of these compression iterations, a place is assigned a specified number of chunks (line 2), while the local `DistArray` objects are reused. This method makes our algorithm suitable for nodes with various memory sizes, provided the chunks are small enough. Note that the chunks can be made smaller by simply dividing the input data set into more chunks. It is expected that too many such chunks would lead to a decrease in performance, as there would be redundant filter and push operations for the same terms at the same place in different loops. We assess this trade-off through the evaluation in Section 5.4.3.

5.3.3 Transactional Data Processing

A commonly occurring scenario is real-time processing of RDF data sets. In such cases, data is inserted as part of a *transaction*, and normally the chunks of data inserted are very small containing only a few hundred statements. In such a scenario, there is no need to distribute data sets. Instead, one could just compress the data set using a single cluster node. In our prototype, the number of cluster nodes is controlled by the `X10_NPLACES` option. Furthermore, parallel transactions with multiple data sets on multiple nodes are also supported using the same option. Finally, an optimized data-node assignment strategy can be integrated with our implementation if needed, but such a strategy is out of the scope of this paper. Similarly, in this paper, we do not address rolling back transactions or deletes. In general, although our system can be extended to support transactional loads, its main utility is in encoding large datasets.

Algorithm 8 Processing Data Chunks in Loops

1: **for** $i \leftarrow 0..(loop-1)$ **do**
2: Assign each place c data chunks
3: Parallel processing at each place
4: **end for**

5.3.4 Incremental Update

Another typical application is the incremental update of RDF data sets. It is often required that such systems must encode a new dataset as an increment to already encoded datasets. Typically, the new input data set is large. In this scenario, local dictionaries could be read in memory before the encoding process. The extension of our algorithms for incremental update is shown in Algorithm 9.

Algorithm 9 Processing Update

1: `finish async at` $p \in P$ {
2: **for** $<key,id> \in local\_dict$ **do**
3: $table(here.id).add(key,id)$
4: **end for**
5: Processing new data
6: }

5.3.5 Algorithmic Complexity

Our compression algorithm with the aforementioned improvements has a worst case computational complexity linear in the number of statements of the input datasets $O(|N|)$ and the number of places $O(|P|)$. Herein, we describe the formulation of our worst case complexity.

For a given place, the worst case complexity of the algorithm is $|P|$, where $|P|$ is the number of places. This complexity is determined by the largest loop at line 13 in Algorithm 5. The total complexity of the algorithm is $O(|P| \times |P| \times |loop|/|P|)$, because there are a total of $|P|$ places and all their implementations are nested inside the `loop` variable in Algorithm 8. The divisor ($|P|$) arises because each of these loops run in parallel. Therefore, the overall worst case complexity is ($O(|loop| \times |P|)$). Based on this, (1) For a constant number of places, the complexity of the algorithm is: $O(|loop|)$, hence, the complexity of the algorithm is linear in the value of loop. Moreover, if the size of each chunk is fixed, assuming `k` triples per chunk and the total number of triples are `N`, then the `loop` would be ($|N|/|k|/|P|$). Thus, the complexity of the algorithm will be $O(N)$, namely linear with the number of input triples `N`. (2) Similarly, for a constant input size, the complexity of the algorithm will be $O(P)$ linear in the number of places or cores in the underlying execution architecture, provided each logical place is mapped to a single core (as in our case).

5.4 Evaluation

We have conducted a rigorous quantitative evaluation of the proposed encoding algorithm. We divide the presentation of our evaluation into different sections. Section 5.4.1 describe the experimental setups. Section 5.4.2, compares the runtime and compression performance of our algorithm against the MapReduce implementation [116]. We also evaluate the runtime performance of our algorithm for the transactional and incremental update scenarios as described previously. Section 5.4.3 examines the scalability of our algorithm and compares it against the scalability achieved by the MapReduce approach for increasing both numbers of processing units and input data set size. Finally, we present the load-balancing characteristics of our system in Section 5.4.4.

5.4.1 Experimental setup

Platform. Our evaluation platform was the *High Performance Systems Research Cluster* in IBM Research Ireland. Each computation unit of this cluster is an iDataPlex node with 2 Intel Xeon X5679 processors each with 6 hardware cores running at 2.93 GHz, resulting

in a total of 12 cores per physical node. Each node has 128GB of RAM and a single 1TB SATA hard-drive. Nodes are connected by Gigabit Ethernet switch. The operating system is Linux kernel version 2.6.32-220 and the software stack consists of Java version 1.6.0_25 and gcc version 4.4.6.

Setup. We have used X10 version 2.3 compiled to C++ code. We set the `X10_NPLACES` to the number of cores and the `X10_NTHREADS` to 1, namely, one activity per place, which avoids the overhead of context switching at runtime.

We compare our results with the MapReduce compression programme [116]. We use the latest version and run it on Hadoop v0.20.2. We set the following system parameters: *map.tasks.maximum* and *reduce.tasks.maximum* to 12, the *mapred.child.java.opts* to 2 GB and the rest of the parameters are left to the default values. The implementation parameters are configured with the recommended values: *samplingPercentage* is set to 10, *samplingThreshold* to 50000 and *reducetasks* to the number of cores. We have verified the suitability of these settings with the authors (of [116]).

We empty the file system cache between tests to minimize the effects of caching by the operating system and run the test three times, recording average values.

Datasets. For the evaluation, we have used a set of real-world and benchmark datasets (as Table 5.1): DBpedia [12], LUBM [53], BTC2011, Uniprot [10]. We chose these data sets because they vary widely in terms of size and kind of data they represent, as described in Chapter 2. The popularity and diversity of these datasets contributes to an unbiased evaluation.

5.4.2 Runtime

Data Compression. We perform the encoding using 16 nodes (192 cores) and report the compression results achieved by our algorithm in Table 5.1: Column *# Stats* gives the number of statements (triples) in each benchmark. The size of the input data sets is given both in the terms of plain and gzip format in columns 3 and 4. The output column is composed of the compressed statements and the corresponding dictionary tables at all places. Finally, the resulting compression ratio is calculated by dividing the size of the input files (in plain format) by the size of the total output. The compression ratios for the four data sets are similar: in the range of $4.1 - 4.5$. Note that although these ratios are smaller than the compression ratio achieved by `gzip`, our output data can be processed directly and we can also compress these outputs further using `gzip`, if need be. We achieve smaller compression ratios compared to MapReduce [116], because we use 64-bit integers to encode all terms,

while their approach uses smaller integers for encoding parts of terms as well as further `gzip` compression on their output data.

Table 5.1 Dataset information and compression achieved

Dataset	# Stats.	Input (GB)		Output (GB)		Compr.
		Plain	Gzip	Data	Dict.	Ratio
DBpedia	153M	25.1	3.5	3.5	2.7	4.1
LUBM	1.1B	190	5.5	24.8	17.7	4.5
BTC2011	2.2B	450	20.9	65.6	40	4.3
Uniprot	6.1B	797	58.7	136	46.4	4.4

Runtime and Throughput. We compare the runtime and throughput between our approach and that of the MapReduce framework in two cases: disk-based and in-memory compression. In the first case, the reading and writing data is on disk (or HDFS based on disk). For the latter, we process all data in memory. For memory based I/O, we pre-read the statements in an `ArrayList` at each place and also assign the output to `ArrayList`. As MapReduce does not provide such mechanisms, we instead set the path of the Hadoop parameter *hadoop.tmp.dir* to a `tmpfs` file system resident in memory. The results of these two cases are shown in Table 5.2 and Table 5.3. We define runtime as the time taken for the whole encoding process: reading files, performing encoding and writing out the compressed triples and dictionaries. The throughput is described in terms of two aspects: (a) rate, which is calculated by dividing the input size (in plain format) by the algorithm runtime, and (b) statements processed per second that is calculated by dividing the number of processed statements by the runtime.

From Table 5.2, our approach is $2.9-7.3\times$ faster than the MapReduce-based approach for disk-based computation, and $2.6-7.4\times$ for in-memory as illustrated in Table 5.3. The smallest speedup occurs for the BTC2011 benchmark, however it should be noted that in this instance, whereas we compress `N-Quads`, MapReduce discards the fourth term in the input data and just compresses the first three terms. Moreover, the compression throughput of Uniprot in both cases is much higher than the other three datasets. We attribute this to the large number of recurring popular terms. Comparing the two cases, the in-memory compression is faster than the disk-based one for both algorithms, although not dramatically so. Moreover, the improvements we achieved in Table 5.3 are greater than those in Table 5.2 for the LUBM and Uniprot data sets, marginally greater for DBpedia and slightly smaller for the BTC2011 data set. This illustrates that the two algorithms gain disproportionally from the faster I/O over different data sets (with our system showing better gains overall).

Table 5.2 Disk-based runtime and rates of compression (192 cores)

Dataset	Runtime (sec.)		Rates (MB/s)		Imprv.
	MapR.	X10	MapR.	X10	
DBpedia	430	59	59.7	435	7.3
LUBM	1739	453	111.9	429.5	3.8
BTC2011	2817	956	163.6	482	2.9
Uniprot	6160	1515	132.5	538.7	4.0

Table 5.3 In-memory runtime and rates of compression (192 cores)

Dataset	Runtime (sec.)		Rates (MB/s)		Imprv.
	MapR.	X10	MapR.	X10	
DBpedia	368	50	69.8	514	7.4
LUBM	1382	254	140.8	766	5.4
BTC2011	1809	708	254.7	650.8	2.6
Uniprot	5076	937	160.8	871	5.4

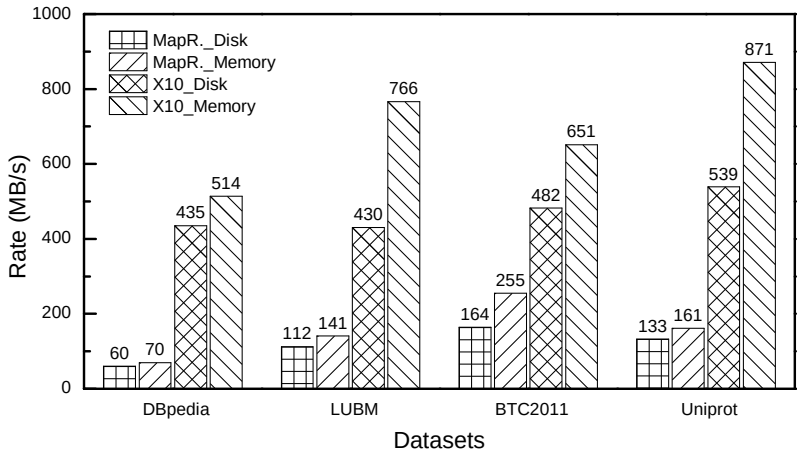


Fig. 5.2 Throughput of the two implementations using 192 cores, based on disk-based and memory-based cases with the four datasets.

Moreover, Figure 5.2 shows that the maximum number of statements processed per second is about 6.51M, higher than any method in the literature.

Transactional. We simulated two transactional processing scenarios with in-memory compression: (1) sequential transactions on a single node and (2) multiple parallel transactions on multiple nodes using the LUBM data set. To simulate transactions, we first encode the 1.1 billion triples in the *LUBM8000* benchmark. Next, we prepare a RDF data set that contains 1M triples, split into 10K, 1K, 100, and 10 chunks, respectively. After encoding is

complete, we encode these new input chunks (every 10 chunks) sequentially and record the corresponding encoding time. For the multiple parallel transaction scenario, we could only record the encoding time for our implementation since Hadoop uses a centralized model for data storage.

Results are presented in Table 5.4. One can clearly observe that our approach is orders of magnitude faster than the MapReduce approach for the sequential case. The latter is neither optimized nor suitable for this use-case, since the startup overhead dominates the runtime, as evident from the observation that the average time to process chunks with different sizes is approximately the same. For our system, we observe that the average runtime of our approach increases with increasing chunk sizes, and the trend moves toward linear for the sequential case. This means that, for a single place, overhead takes a larger proportion of the runtime.

Table 5.4 Processing 1M statements in the transactional scenario

# Stats	***Avg. runtime per 10 chunks (sec.)***		
per chunk	**MapR.**	**X10**	**X10_Para.**
100	439	0.211	0.164
1K	441	0.359	0.391
10K	454	1.761	0.648
100K	454	17.177	2.192

Since we are using 192 cores and the number of chunks used in this scenario is 10, for each transaction with the parallel processing by our prototype, the chunks can be compressed at once by 10 places in parallel. The results in Table 5.4 show that the runtime is around 0.2 seconds when the number of statements is less than 100 in each chunk, which is slightly worse than our expectations for real-time applications, although still well within an acceptable range. Upon further analysis, we have found that this increase in program runtime is due to underlying bottlenecks in the X10 runtime implementation, which we have not addressed in this thesis: (a) Every `async` call forks an underlying `pthread` (Posix thread) *atomically*, which leads to execution time overhead. (b) Type initializations in X10 are expensive, because all type initializations are internally guarded by locks. Our implementation still performs reasonably well even with these implementation overheads.

Updates. We evaluate the incremental updates scenario for RDF compression using the *LUBM8000* dataset and by splitting it into 2, 4, and 8 chunks, respectively. The resulting datasets are compressed in 2, 4 and 8 different executions respectively. Before each compression cycle, we empty the cache as to simulate real world conditions. The results

Table 5.5 Incremental update scenario with different chunk size

# Chunks	Chunk Size	Runtime (sec.) MapR.	Runtime (sec.) X10	Imprv.
1	190 GB	1739	453	3.8
2	95 GB	2468	551	4.5
4	47 GB	3900	755	5.2
8	23 GB	6704	1164	5.8

comparing our approach and MapReduce are shown in Table 5.5. As expected, the performance for both algorithms decreases with increasing number of chunks, because of the additional process required during the encoding (e.g. reading the dictionary into memory). However, the increase in program runtime for our approach is much smaller than MapReduce. A possible explanation is that because our dictionary reading operation is faster, the startup overhead of our system is lower. It is also possible that the efficacy of the popularity caching technique used by MapReduce decreases disproportionately as the number of chunks increases.

5.4.3 Scalability

We test the scalability of our algorithm by varying the number of processing cores and the size of the input data set. We use the LUBM benchmark in our tests as it facilitates the generation of datasets of arbitrary size.

Number of Cores. We fix the input data set to 1.1 billion triples and double the number of cores from 12 (single node) till 384. The test results for our algorithm and the MapReduce-based approach are shown in Figure 5.3. These results demonstrate that the run time for both algorithms decreases with an increase in the number of cores.

The speedup obtained with an increasing number of cores compared to a baseline of 12-cores for both algorithms is presented in Figure 5.4. In our system, with a small number of cores, the runtime is not linear, since for a single node there is no network communication. Nevertheless, starting from 24 cores, the speedup becomes almost linear (scaled speedup, not shown in the figure, is approximately 1.95). This result supports our theoretical analysis in Section 5.3.5, and we attribute the small amount of loss to network traffic. In contrast, the speedup of the MapReduce-based approach is almost linear (or even better) initially before plateauing for values of 92 cores and greater. This result mirrors the result obtained by Urbani at al. [116]. There can be several reasons for the latter slowdown: we hypothesize

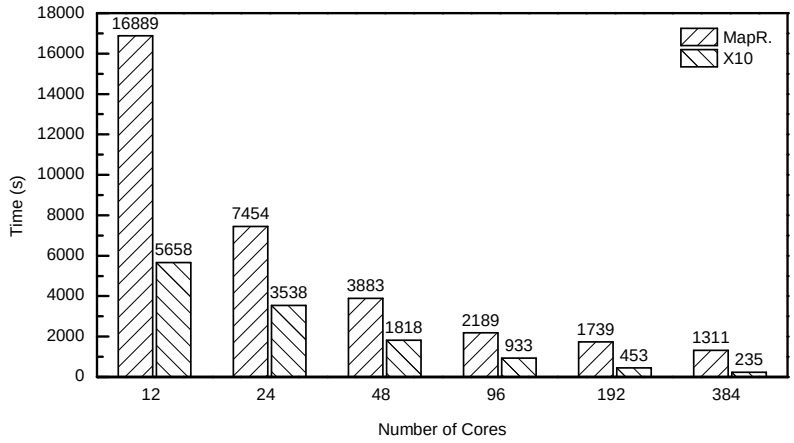


Fig. 5.3 Runtime by varying nodes.

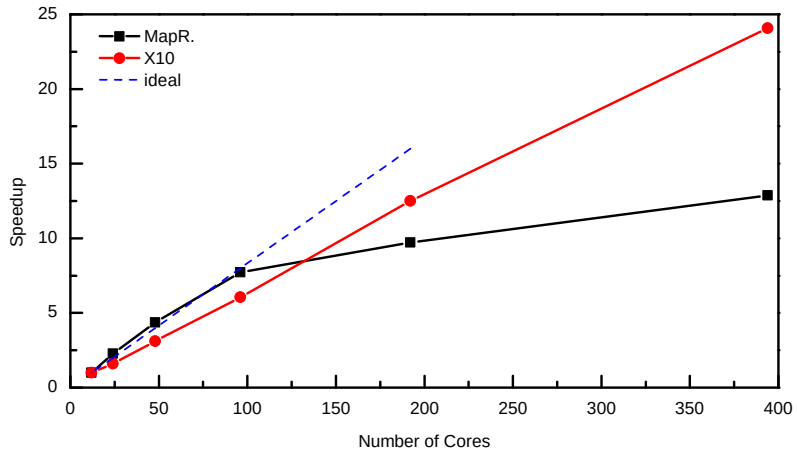


Fig. 5.4 Speedups by varying nodes.

that this may be due to load imbalance, increased I/O traffic and platform overhead.

Size of Datasets. To study the scalability of our algorithm with increasing input data size, we create a large LUBM data set with 11 billion triples, which is roughly equivalent to the *LUBM80000* benchmark. We split this data set into a number of chunks, each of which contains 140K triples, al•lowing us to study the effect of *loop* from Algorithm 8.

We start our tests with 690 million triples and repeatedly double the size of the input until we reach a dataset comprising 11 billion triples. Additionally, for each dataset, we also vary the number of *chunks read per loop* for our implementation. The results are presented in Figure 5.5. We see that the runtime for both algorithms is nearly linear with the size of the input data sets. We also notice that MapReduce achieves a slightly super-linear speedup until 5.5 billion triples. After that, MapReduce speedup becomes linear with the input size. For our algorithm, we have experimented with 1, 5, and 10 chunks in each loop.

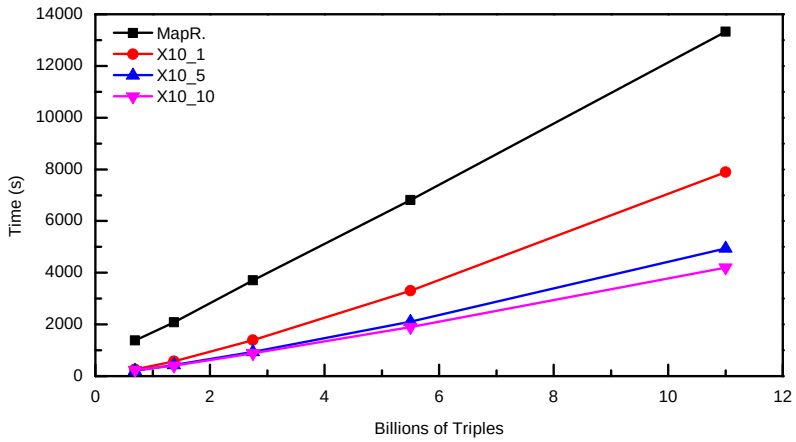


Fig. 5.5 Runtime by varying size.

One can see that the scalability of our algorithm is not linear with input data when reading 1 chunk per loop. But, speedup becomes better as we increase the number of chunks read per loop, and it matches the ideal linear speedup scenario when reading 10 chunks per loop. The reason may be the same as for the transactional case mentioned above, i.e. that a large number for *loop* results in additional runtime overheads as a result of forking threads and object type initializations. Small chunks also results in redundant *filter* and *push* operations for the same terms at the same place in different loops. Such an interpretation is in sympathy with our expectations described in Section 5.3.2.

Furthermore, Figure 5.5 investigates the trade-off between reduced memory consumption and performance as well. For the optimal scalability case with reading 10 chunks at a time, we need to process $10 \times 140K = 1.4M$ triples in each loop. Since, in Table 5.1, we show that 1.1 billion triples is about 190 GB, the size of 1.4 million triples would be about 250 MB, which is well within the RAM availability of most machines. Not withstanding this optimal case implementations using 5 chunks at a time (125 MB) and 1 chunk at a time (25 MB) is only accompanied with little and moderate scalability loss respectively.

5.4.4 Load Balancing

We measure the load-balance characteristics of our algorithm in terms of five metrics defined later in this section. We instrument our code with counters to gather data for the first four metrics. The data for the final metric is obtained using the tracing option provided by the X10 implementation.

- *number of outgoing terms*: The number of terms transferred to a remote place. This

metric gives insight into the *communication* load balance achieved by our algorithm. For example, the larger the number of outgoing terms, the greater the associated network traffic.

- *number of misses*: The number of terms that are not already encoded (*missed*) in the dictionary and hence require the generation of a new id.
- *miss ratio*: The number of misses divided by the sum of hit and miss for the local dictionary.
- *number of processed terms*: the number of terms processed by a computing node.
- *received bytes*: the size of processed terms in bytes at a computing node.

We encoded 1.1 billion LUBM triples on a varying number of cores to gather data for the first three metrics described above. The results are presented in Table 5.6. We can see that the average values of the three metrics for all the tests are very close to the maximum values, suggesting excellent load balancing performance. The scalability of our algorithm with an increasing number of processing cores is highlighted well in these results. There is a clear linear decrease in all three metrics with an increase in the number of processing cores. Finally, the results also illustrate a consistent almost uniform miss probability for each dictionary. The average miss ratio is about 94.5%, indicating that we have redundant computation on average for 5 out of every 100 terms. This ratio approached the ideal value of 100%, which is nevertheless difficult to achieve in a distributed systems without significant coordination overhead. Additionally, our implementation is still based on the *all-to-all* communication, which could possibly affect the performance. However, our system does not repartition all the data, but only transfers the mappings that are necessary for each node. In this sense, our system performs useful computation in terms of data locality in 94.5% of the cases, meaning that although our approach does require communication between all nodes, only moving the data when actually needed.

The last two metrics capture the load at each compute node in terms of the number of terms processed and size of data received in bytes. These metrics are important for measuring computational load balance and are used here to provide comparison with the performance available using the MapReduce approach. Since MapReduce divides the whole compression into three separate jobs and the implementation does not provide the relative metrics, we extract the *reduce input records* and *reduce shuffle bytes* in the reduce phase of each job from the Hadoop logs. These two items indicate the number of records processed and the corresponding data sizes for each of the 192 reduce tasks.

Table 5.6 Detailed term information during encoding 1.1 billion triples

# Core	# Outgoing (M)		# Misses (M)		Miss Ratio	
	Max	Avg.	Max	Avg.	Max	Avg.
24	11.65	11.59	10.95	10.95	95.7%	94.5%
48	5.85	5.78	5.46	5.46	96.1%	94.5%
96	2.94	2.89	2.73	2.73	96.1%	94.5%
192	1.48	1.43	1.35	1.35	96.4%	94.5%
384	0.74	0.70	0.90	0.87	96.4%	94.5%

Table 5.7 Comparison of received data for each computing node when processing 1.1 billion triples using 192 cores (in millions)

Algorithm		Recv. Bytes		Recv. Records	
		Max.	Avg.	Max.	Avg.
MapR.	Job1	9.94	4.02	24.04	1.73
	Job2	135.61	79.77	30.91	17.28
	Job3	120.81	106.82	19.61	17.28
X10		194.71	187.82	1.48	1.43

The results are summarized in Table 5.7 and demonstrate that the difference between the maximum and the average value of these metrics for our implementation is much smaller than MapReduce, indicating better load balancing (in addition to the results, the minimum number of bytes received is $184.70M$ and the minimum number of records received is $1.37M$ in our approach, also showing minimal skew). Furthermore, when comparing the sum total of bytes received across the two implementations, it is clear that our proposed technique results in better performance. Consequently even when comparing with the reduce phase of MapReduce, our system results in a lighter workload and less network communication, especially taking into consideration that we are using a longer representation (64 bits).

5.5 Discussion

Based on the above results, we can see that our proposed implementation is highly efficient and much faster than the state-of-art method [116]. It should be highlighted: though the presented implementation is based on the X10 parallel language, our proposed method actually can be easily implemented by any other modern parallel language used in *high performance computing* such as MPI etc. In the meantime, compared with MapReduce

method [116], our algorithm has the following two obvious differences, which make our method have more notable advantages in both computation and network communications when processing large-scale RDF data.

1. We do not quantify any skew. We just employ a *filter* structure (for example a simple `hashset`) to process the terms and to extract the unique terms that need to be transferred to the remote node. This is done for all terms irrespective of their popularity. Using the filter guarantees that any given term can possibly move to a remote node just once per current node, which is shown to be very efficient for handling the data skew existing in the semantic web in our evaluations in Section 5.4.

2. We only need to send unique terms to remote dictionaries and retrieve their ids, but not transfer any triples at all. In comparison, the approach [116] has to follow the MapReduce model strictly. It has to decompose all the triples in the form of <*key,value*> pairs and redistribute all of them among all the nodes. Furthermore, all the terms have to be redistributed again after the encoding process so as to reconstruct all the id triples. This could bring very heavy network communication and also computations, impacting the encoding performance.

5.6 Conclusions

In this chapter, we have introduced an efficient dictionary encoding algorithm for the compression of big RDF data. We have presented an extensive quantitative evaluation of the proposed algorithm and conducted a comparison with a state-of-art system using the MapReduce model[116]. Our main conclusions are that the proposed algorithm is: (1) Highly scalable both with increments in number of cores and in the size of the dataset, (2) Computationally fast, encoding 11 billion statements in about 1.2 hours, and achieving a $2.6-7.4\times$ improvement over the MapReduce method, (3) Flexible for various semantic application scenarios, (4) Robust against data skew, showing excellent load balancing, and (5) Suitable for use and further development as part of a high performance distributed system.

We will build efficient indexes for the encoded triples in Chapter 8. Before that, we will focus on improving the join (both the inner- and outer joins) performance for our framework in the following two chapters.

Chapter 6

A Novel Framework for Handling Skew in Parallel Joins on Distributed Systems

6.1 Introduction

Following the three operations as described in our proposed framework, we now investigate efficient parallel join methods over large-scale data in this chapter. More specially, since data skews exist naturally in various applications, we focus on efficient skew handling techniques in join implementations.

As we have stated in the related work in Chapter 2, although different techniques and algorithms have been proposed to handle skew in joins, all of them so far still rely on the conventional frameworks already described, namely the hash-based and duplication-based joins. In contrast, in this chapter, we propose a novel framework as an alternative to the conventional approaches, called *query-based distributed join*, for efficiently handling data skew in massively parallel joins on distributed systems. From this basis, we also propose a new method called *query with counters* (QC), for directly and efficiently processing skews in parallel outer joins.

For both the inner- and outer joins, we develop efficient distributed algorithms and implement our parallel joins using the X10 language [22]. We evaluate performance on an experimental configuration consisting of 192 cores (16 nodes) and large datasets of 1 billion tuples with different skews. Moreover, we also compare our approaches with the state-of-art methods and show experimentally that our algorithms performs faster in the presence of high skew.

The rest of this chapter is organized as follows: In Section 6.2, we present our *query-based distributed join* framework. In Section 6.3 we apply the framework to outer joins.

The detailed implementation of our proposed approaches are presented in Section 6.4. We provide a quantitative evaluation of our inner join algorithms in Section 6.5 and the outer joins in Section 6.6. We conclude the work in Section 6.7.

6.2 Query-based Distributed Join

In this section, we first introduce our *query-based distributed join* framework and its detailed work flow. Then we analyze how this scheme can efficiently handle data skew. Furthermore, we also discuss its advantages and disadvantages compared with current approaches.

6.2.1 Framework

Assuming the input relations are R and S, where $|R| < |S|$ and S is skew, there are N computing nodes, and before the join operations the ith node has a subset of both relations R_i and S_i. As shown in Figure 6.1, our framework has two different communication patterns - distribution and query, between local and remote nodes, which obviously makes it different from the conventional hash-based and duplication-based frameworks. Here, we divide its detailed work flow into the following four phases.

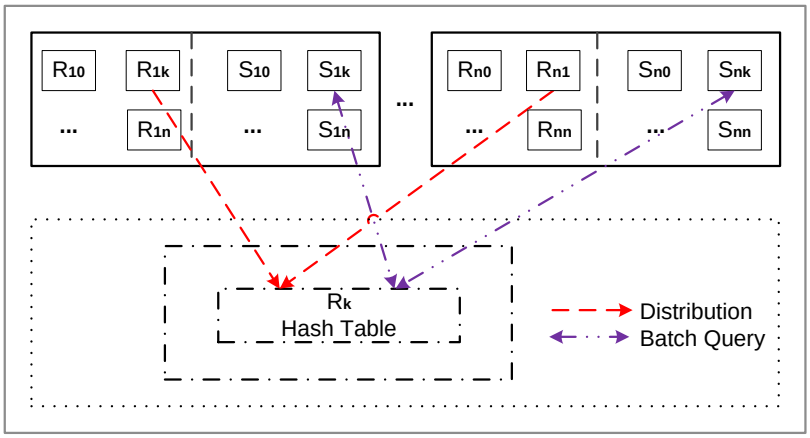


Fig. 6.1 Query-based Distributed Join Framework. The dashed rectangle refers to the remote computation nodes and objects.

R Distribution

The relation R is processed in the same way as the hash-based implementations, in that each R_i is partitioned into N chunks, and each tuple is assigned according to the hash value of its

key by a hash function $h_1(k) = k \ mod \ N$. After that, all the chunks R_{ij} will be transferred to the jth node. There are two reasons to do so: (1) R is relatively small such that we can afford the distribution cost, and (2) R can be considered as a uniform distributed data set, as adding skew to the relation R would violate the primary key constraint [16].

Push Query Keys

In this phase, we scan each tuple in the relation S at each node and insert them in a set of local hash tables T_i (the number of hash tables is N). The tuple assignment is according to $h_1(k) = k \ mod \ N$ as well, such that the tuples having the hash value j are put into the jth hash table T_{ij}. The structure of the hash tables is shown as Figure 6.2(a). It supports the $1 \rightarrow n$ mappings, such that tuples with the same keys will be stored in the same bucket. After that, iterations on each hash table commence and all keys in each hash table are picked up and kept sequentially in memory. Finally, we push the keys from the hash table T_{ij} to the j node, where these keys are called the *query keys* of the node j in our approach.

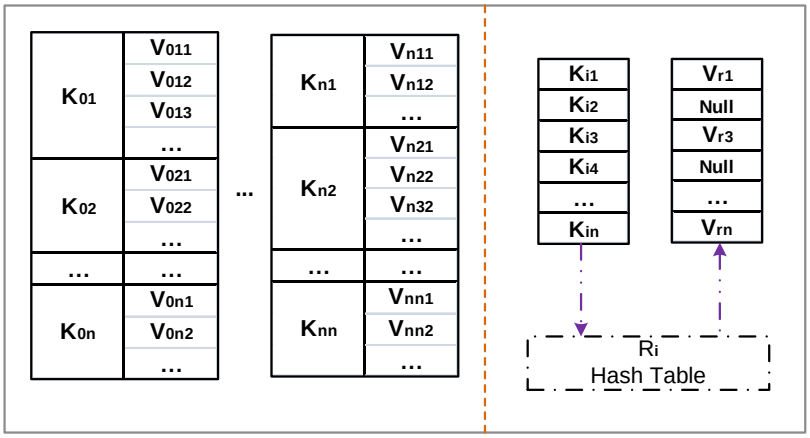


Fig. 6.2 The data structure used in query-based distributed join: (a) the local hash tables of S (left), and (b) the query keys of a remote node and its corresponding returned values (right).

Return Queried Values

In this step, we first build a local hash table T_i' at each node, based on the received tuples from the first phase. After that, we look up each of the received query keys in T_i' and output the matched values. If there is no matching keys, the value will be set to `Null`. All these values are also kept sequentially as well as the corresponding query keys. This process can be seen in Figure 6.2(b), where all the values are called *returned values*, because we push

these values back to the nodes where the query keys originally come from after finishing the lookups.

Result Lookups

After receiving sets of returned values from remote nodes, we start to scan these values at each node. Take a node i for example, for the returned values from jth node, we first check whether the value is null. If the value is null, we continue scanning the next value. If it is not, it means that there is a match between R and S. The reason is that each query key is extracted from S, and a non-null returned value means that this key exists in R as well. Therefore, we look up the corresponding query key in the corresponding hash table T_{ij} and output the join results. The join operation ends with the output of all the results.

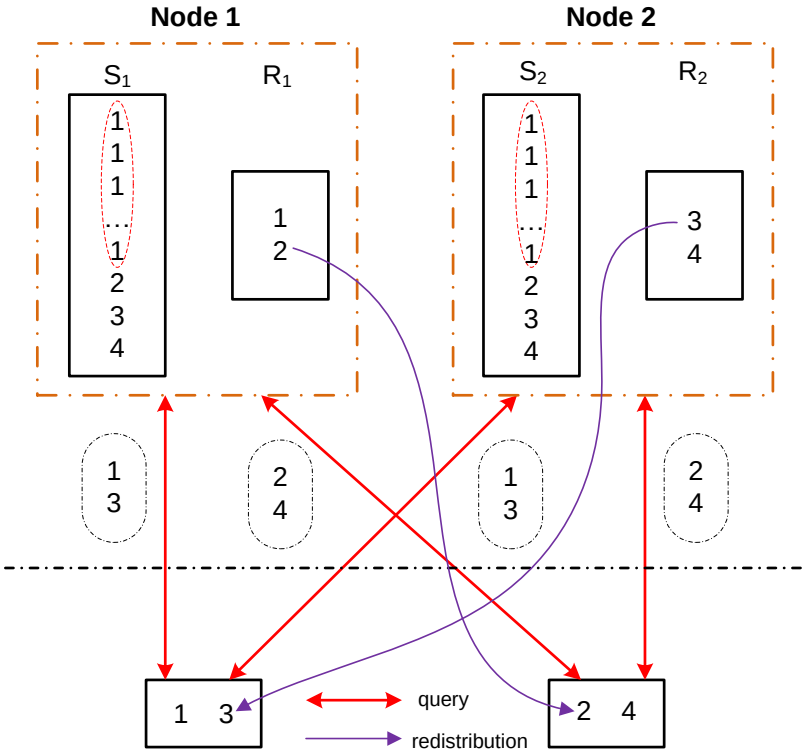


Fig. 6.3 An example of the query-based implementation over a two-node system.

6.2.2 Handling Data Skew

We refer to our algorithm as *query-based* because the process of transferring keys to remote nodes and retrieving the corresponding values looks like a *query*. It can be seen that, though S is skewed, we do not transfer any tuples of this relation in our framework. Instead, we just

transfer the keys of S. More exactly, we only distribute the **unique** keys of S on the basis of $1 \rightarrow n$ structure of hash tables T_i. An example of such implementation is demonstrated in Figure 6.3. It can be seen that after the redistribution of the relation R, the relation S_1 and S_2 at the two nodes only needs to *query* the key set {1,3} and {2,4} to the responsible remote node to retrieve their values.

For a common case, assuming that there exists skew tuples, which have the same key k_s, and this key appears n_s (large number) times in the relation S. Using the conventional hash-based method, all these n_s tuples will be transferred to the $h_1(k_s)$-th node, which results in a hot spot both in communication and in the following probing operations. By comparison, our framework efficiently addresses this problem in two aspects: (1) each node will receive **only** one key (or maximum N keys if these tuples are distributed on the N nodes), and (2) each query key is treated as the same in the following look up operations.

6.2.3 Comparison with other Approaches

In addition to efficient handling of data skew, compared with the conventional frameworks, our scheme still has two other advantages: (1) network communication can be highly reduced, because we only transferred parts of keys in S, and their corresponding returned values, and (2) computation can be decreased when S is high skew, because (a) though we have two lookup operations on T_i and T_i', the hash tables in T_i will be very small, (b) skew tuples will be looked up only once instead of checking all of them and (c) lookup operations for the tuples that are not participating in the join results are removed by just checking whether the returned value is null or not.

Taking a higher level comparison with the histograms [6] and the PRPD [127] methods as described previously, there are two other advantages to our approach: (1) we do not need any global knowledge of the relations in the presence of skew while [6] and [127] require a global statistic to quantify the skew, and (2) our approach does not involve redundancy of join (or lookup) operations while the other two have, because each node in our method is just *query what I need*, while [6] and [127] have *broadcast behavior*, such that some nodes may receive some tuples what they do not really need.

In our framework, we have to build local hash tables for S_i at each node, which could be time-costly. Additionally, when the skew is low, the number of query keys will be uncompetitive as well, and the two-sided communication will decrease the performance. We assess the balance of these advantages and disadvantages through evaluation with real-world datasets and an appropriate parallel implementation in Section 6.5.

6.3 Applying to Outer Joins

The main difference between the implementation of inner joins and outer joins is to distinguish the matched and non-matched tuples over a distributed system. From the basis of the proposed join framework, in this section, we show how we use a simple structure to seamlessly realize such function for the outer joins.

The new approach is demonstrated in Figure 6.4. We can see that the used general communication patterns are the same as the query-based framework. The only difference is that a data structure named *query counter* has been integrated, and that is the reason why we call this method *query with counters*. Similar to the process described for the inner join implementation, we also divide the detailed work flow into four steps. Since the first two steps *R distribution* and *push query keys* are the same as the query-based joins, we only described the third and the fourth steps.

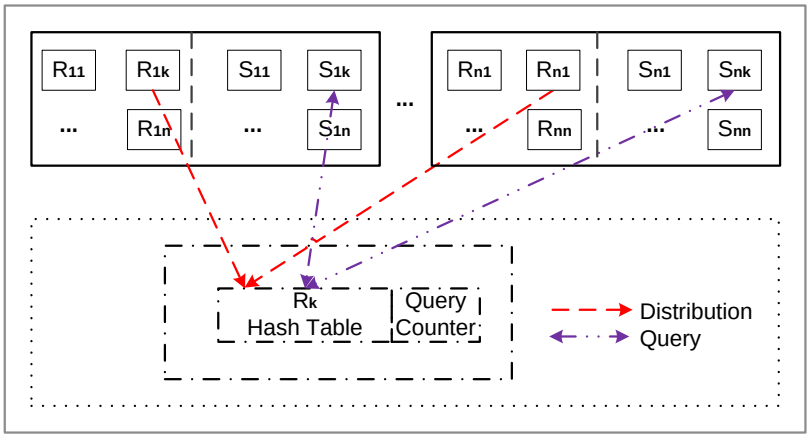


Fig. 6.4 The Query with Counters approach for outer joins. The dashed rectangle refers to the remote computation nodes and objects.

We call the third step *count matches and return queried values*. In this step, we first build a local hash table T_i' with the data structure <*key, (value, counter)*> at each node, in which the *key* and *value* are the received tuples from the first phase while the *counter* is an integer and initialized as 0. After that, we look up each of the received query keys in T_i' and output either a matched value or `Null`. The same as the inner joins, all these values are kept sequentially as the corresponding query keys and pushed back to the nodes where the query keys originally come from after finishing the lookups. The detailed process can be seen in Figure 6.5. If a match exists, the returned value will be the matched value, meanwhile, we also increase the corresponding *counter* by one. If there is no match in R_i, the returned value will be set to `Null`, and there is no operation for the *counters*.

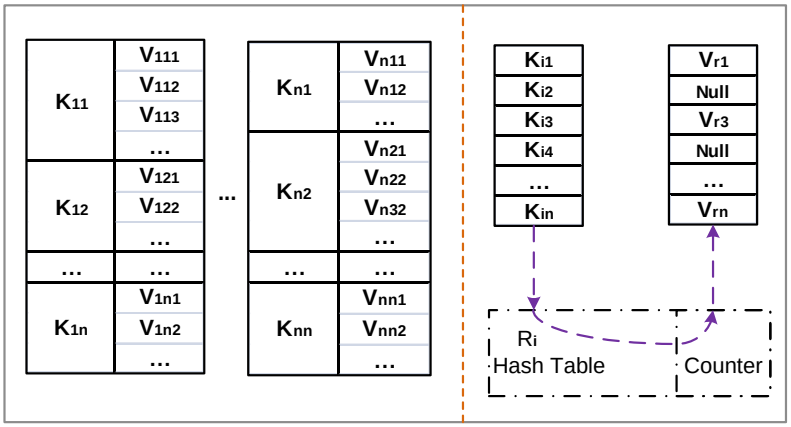


Fig. 6.5 The data structure used in QC algorithm: (a) the local hash tables of S (left), and (b) the query keys of a remote node and its corresponding returned values (right).

The fourth step is also the *result lookup*. After receiving sets of returned values from remote nodes, we can formulate the final join results. We divide this process into two kinds of lookup: (1) Matched result lookup, which is the same as the inner joins, through scanning the received values at each node and lookup the responsible local hash tables of T. (2) Non-matched result lookup, through checking whether the *counter* is 0. We iterate all the keys in the hash table T' and check the corresponding *counter*. For each $counter = 0$, we output the non-matched result of the corresponding key directly. The reasons are: (1) the query is based on the hash-based implementation, and (2) the key in R with $counter = 0$ means that this key has never been matched with the query keys, and also means it has no match in S. The join operation ends with the output of all the results.

It is obvious that this new algorithm inherits the skew handling advantages from the query-based framework and that only the unique keys of the relation S are transferred regardless of their popularity. Moreover, by using a local query counter, we can directly identify the non-matched results while the described methods [6] and [127] needs more complex pre-distribution or redistribution operations. In the meantime, although the presented DER [126] algorithm has done specified optimization for the inner implementation of outer joins, it still needs to redistribute the row-ids. All of these highlight that our approach is more straightforward on processing outer joins. We will evaluate its performance and compare it with the state-of-art techniques in Section 6.6.

Additionally, the QC join approach we use here can also be easily applied to other kinds of joins directly. For example, the returned *null* can be applied directly for right outer-joins and the *counters* for anti-joins etc.

6.4 Implementation

As the QC algorithm is mainly based on the proposed *query-based join* framework, in this section, we only present the detailed implementation of the basic *query-based* method using the X10 framework (detailed QC implementation in X10 is given in Appendix A). We compare our algorithms with the state-of-art PRPD algorithm [127] and PRPD+DER [127] [126] respectively. Since they do not provide any code-level information, we have also implemented them in X10.

6.4.1 Parallel Join Processing

R Distribution. We are interested in high performance distributed memory join algorithms, therefore, we first read all the tuples in `ArrayList` at each node, and then start to distribute the relation R. The pseudocode of this process is given in Algorithm 10. The array R_c is used to collect the grouped tuples, and its size is initialized to the number of computing nodes N. Then, each thread reads the arraylist of R and groups the tuples according to the hash values of their keys. After that, the grouped items are serialized and sent to the corresponding remote place. This process is done in parallel, and we use the `finish` predicate to guarantee the completion of the tuple transfer in each place before pushing query keys.

Algorithm 10 R Distribution

```
1: finish async at p ∈ P {
2: Initialize R_c:array[array[tuple]](N)
3: for tuple ∈ list_of_R do
4:     des=hash(tuple.key)
5:     R_c(des).add(tuple)
6: end for
7: for i ← 0..(N − 1) do
8:     Serialize R_c(i) to ser_R_c(i)
9:     Push ser_R_c(i) to r_R_c(i)(here) at place i
10: end for
11: }
```

Push Query Keys. The detailed implementation of the second step is given in Algorithm 11. A set of `hashmap` is initialized at each place. Each `hashmap` collects tuples of S according to their hash values. If the key of a tuple has already been in the `hashmap`, then only the value part of the tuple will be added in the hash table. After processing all the tuples, the keys in each hash table will be extracted by an iteration on its `keyset`. These keys will be kept

Algorithm 11 Push Query Keys

```
1: finish async at p ∈ P {
2: Initialize T:array[hashmap[key,ArrayList(value)]](N)
3: for tuple ∈ list_of_S do
4:     des=hash(tuple.key);
5:     if tuple.key ∉ T(des) then
6:         T(des).put(tuple.key,tuple.value)
7:     else
8:         T(des).get(tuple.key).value.add(tuple.value)
9:     end if
10: end for
11: for i ← 0..(N−1) do
12:     Extract keys in T(i) to local_key_c(here)(i)
13:     Serialize local_key_c(here)(i) to ser_key(i)
14:     Push ser_key(i) to remote_key_c(i)(here) at place i
15: end for
16: }
```

in *local_key_c*, and then serialized and pushed to the assigned place for further processing. In this process, both the `array[hashmap]` and *local_key_c* are `DistArray` objects, which are kept in memory for the subsequent result lookups, as mentioned in Section 6.2. We use the *finish* operation in this part to guarantee the completion of the data transfer at each place before the next phase commences.

Return Queried Values. This phase starts after the grouped query keys have been transferred to the appropriate remote places. The implementation at each place is similar to a sequential hash join. The received serialized tuple and key arrays, representing the distributed *R* and grouped query keys respectively, are deserialized. For the tuples, all the <*key,value*> pairs are placed in the local hash table *T'*. The keys are used to access this hash table sequentially to get their values. In this process, if the mapping of a key already exists, its value is retrieved, otherwise, the value will be considered as *null*. In both cases, the value of the query key is added into a temporary array so that it can be sent back to the requester(s). All these processes take place in parallel at each place, and we use the `finish` operation for synchronization. The details of the algorithm are given in Algorithm 12.

Result Lookups. The join results at each place can be looked up after all the values of the query keys have been pushed back. Since the query keys and their respective values are held in order inside arrays, we can easily look up the keys in the corresponding hash tables to organize the join results as shown in Algorithm 13. The entire join process terminates when all individual activities terminate.

Algorithm 12 Return Queried Values

```
1: finish async at p ∈ P {
2: Initialize T':hashmap, value_c:array[value]
3: for i ← 0..(N − 1) do
4:     Deserialize r_R_c(here)(i) to tuples
5:     Put all <tuple.key,tuple.value> into T'
6: end for
7: for i ← 0..(N − 1) do
8:     Deserialize remote_key_c(here)(i) to key_c
9:     for key ∈ key_c do
10:        if key ∈ T' then
11:            value_c.add(T'.get(key).value)
12:        else
13:            value_c.add(null)
14:        end if
15:    end for
16:    Push value_c(i) to r_value_c(i)(here) at place i
17: end for
18: }
```

Algorithm 13 Results Lookups

```
1: finish async at p ∈ P {
2: for i ← 0..(N − 1) do
3:     Deserialize r_value_c(here)(i) to local_value_c
4:     for value ∈ local_value_c do
5:         if value ≠ null then
6:             Look corresponding key in T(i)
7:             Output join results
8:         end if
9:     end for
10: end for
11: }
```

6.4.2 The PRPD-based Methods using X10

For our purposes, the implementations of the PRPD and PRPD+DER algorithm have been described in the previous Chapter 2. As the PRPD needs to partition the tuples according the frequency of their keys at the beginning, we add a *key sampling* process on S to measure the skew, wherein we use a `hashmap` counter with two parameters: (1) *sample rate*, namely the ratio of the tuples to be sampled, and (2) *threshold*, namely the number of occurrences of a key in the sample after which the corresponding tuples are considered as skew tuples.

6.5 Evaluation of Inner Joins

In this section, we present the results of our experimental evaluation for the inner joins on a commodity cluster. We conduct a quantitative evaluation of our implementation and compare them to the results obtained by other algorithms.

6.5.1 Platform

Our evaluation platform is the *High Performance Systems Research Cluster* in IBM Research Ireland. Each computation unit of this cluster is an iDataPlex node with two 6-core Intel Xeon X5679 processors running at 2.93 GHz, resulting in a total of 12 cores per physical node. Each node has 128GB of RAM and a single 1TB SATA hard-drive and nodes are connected by Gigabit Ethernet. The operating system is Linux kernel version 2.6.32-220 and the software stack consists of X10 version 2.3 compiling to C++ and gcc version 4.4.6.

6.5.2 Datasets

The evaluation is implemented on two relations R and S, which are both two-column tables that are populated with random data. The key and payload are both set to 8-byte integers. We fix the cardinality of R to 256 million tuples and S to 1 billion tuples. Join with such characteristics are common in data warehouses and column-oriented architectures.

Three key distributions are examined in our tests: uniform, low skew and high skew. We only add skew to S, following the Zipf distribution. The skew tuples are evenly distributed on each computing node and the skew factor is set to 1 for the low skew (top ten popular keys appear 14% of the time) and 1.4 for the high skew dataset (top ten popular keys appear 68% of the time). Again, highly skewed datasets are very common in a variety of settings in data warehouses and also in non-relational stores (e.g. see [78]).

6.5.3 Setup

We set the `X10_NPLACES` to the number of cores and `N_Thread` to 1, namely one place for one single activity, which avoids the overhead of context switching at runtime. The parameter *sample rate* is set to 10%, and the *threshold* is set to a reasonable number 1000 based on preliminary results. In all experiments, we only count the number of matches, but do not actually output join results. Moreover, we record the mean value based on ten measurements and we empty the file system cache between tests to minimize the effects of caching by the operating system.

6.5.4 Runtime

We examined the runtime of three algorithms: the conventional hash-based algorithm, the PRPD method [127] and our query-based approach. We implement these tests using 16 nodes (192 hardware cores) of the cluster on the datasets with different skews, and present the results in Figure 6.6. We can see that each algorithm has its strengths and weaknesses: (1) when the distribution is uniform, hash and PRPD perform nearly the same and much better than our query-based implementation, (2) with low skew, PRPD becomes the faster with our approach being slightly slower, and (3) with high skew, our approach outperforms the other two and the hash-based implementation shows very poor performance.

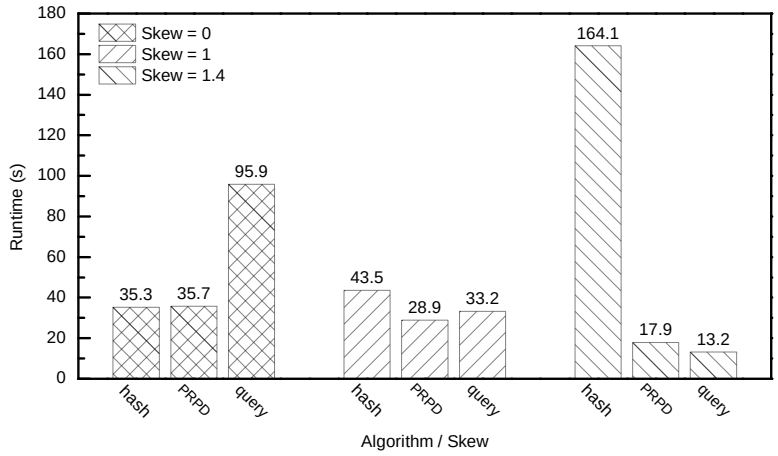


Fig. 6.6 Runtime comparison of the three different algorithms. The join is implemented on $256M \bowtie 1B$ with different skew by using 192 cores.

In the meantime, we also observe that with the increase of the data skew, the time cost of hash method increases sharply while our scheme decreases sharply, which means that our framework has total opposite properties compared with the commonly used hash-based join framework. In the meantime, PRPD is a hybrid method, still in the scope of the conventional approaches, so it has reasonable robustness against skew. Our method performs best under high skew conditions, so our new join framework can be considered as a supplement for the existing schemes. In fact, a system could pick the correct implementation based on the skew or the input so as to minimize runtime.

We have examined the time breakdown on each phase (not shown in the figure) and found that the time cost of our *push query keys* and *return queried values* phase is about three times more than the *S redistribution* and *build & probing* phases of the hash-based implementation respectively. This has corroborated our expectation mentioned in Section 6.2.3.

6.5.5 Network Communication

The number of received tuples (or query keys in our algorithm) for each place indicates both network load and load balancing. As R is uniformly distributed, we only show the part of transferred tuples (keys) of S in each algorithm. We implement our test on 192 cores, and collect the received tuples (keys) at each place by inserting counters. The results of the average number of received tuples for each place in each algorithm is shown in Figure 6.7.

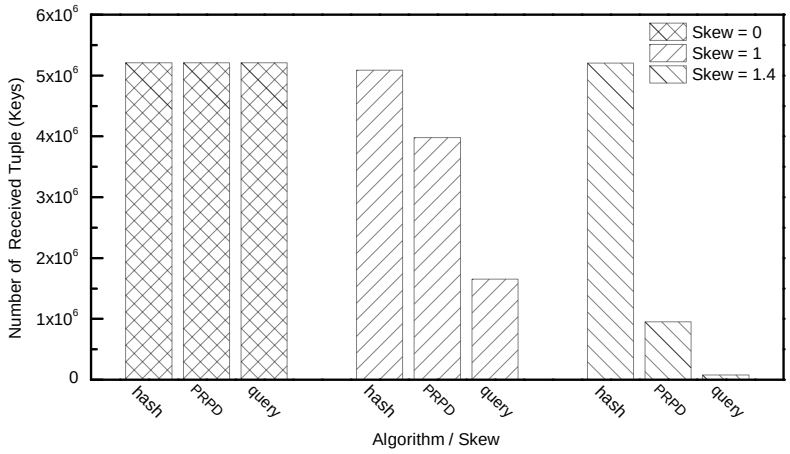


Fig. 6.7 The average number of received tuples (or keys) for each place of the three different algorithms.

We can see that the three algorithms receive the same number of tuples when the dataset is uniform. This is reasonable, since the partial redistribution of PRPD is ineffective as there is no skew and the number of query keys is equal to the number of total keys in our approach. With the increase in skew, the received tuples in the hash-based method does change. In contrast, PRPD and our method show a significant decrease, as they are grouping skewed results more effectively. In addition, our method transfers much less data than PRPD. All of this shows that our implementation can reduce the network communication more efficiently than other approaches under skew.

6.5.6 Load Balancing

We analyze the load balancing of each algorithm based on the metric: *number of received tuples (keys) of S at each place*. We have three reasons to do so: (1) R is uniform distributed, which has no effect for the balance at each place. In the meantime, the broadcast part of R in PRPD does not weaken the balancing as well. (2) The number can indicate the communication and computing time cost, the more tuples (keys) a place receives, the more

time will be spent on data transferring and join (lookup) operation at this place. (3) We have to push the values back and implement the *results lookups* in our query-based algorithm, however, (a) the number of returning values is the same as the received keys, which has the same effect for load balancing, and (b) the final lookups take only a very small part of the whole runtime that can even be neglected.

Table 6.1 The number of received tuples or keys (in millions)

Algo.\Skew	**0**		**1**		**1.4**	
	Max.	**Avg.**	**Max.**	**Avg.**	**Max.**	**Avg.**
hash-based	5.21	5.21	57.68	5.20	324.23	5.21
PRPD	5.21	5.21	6.73	3.98	3.62	0.95
query-based	5.21	5.21	1.68	1.65	0.09	0.08

As the place that receives the maximum number of tuples dominates the final runtime, we just report results of the maximum and average number of the metric, which is shown in Table 6.1. We can see that all three algorithm achieves perfect load balancing when the dataset is uniform. With the skew increase, the load balancing of hash-based algorithm becomes much worse. In the meantime, though PRPD has notable improvement for that condition, our query-based approach is still much better than PRPD, which has nearly not been effected by the data skew.

6.5.7 Scalability

We test the scalability of our implementation by varying the number of processing cores on all three datasets. We start our test with 4 nodes (48 cores), 8 nodes (96 cores) and 16 nodes (192 cores). The detailed time cost of each phase is shown in Figure 6.8.

We can see that the implementation generally scales well with the number of cores. In detail, when the dataset is uniformly distributed, all four phases (referred to as phase 1 etc. according to Section 6.4.1) scale well and the time-cost in the second and third step dominates the whole performance. When the distribution is skewed, we observe that phase 1 and phase 2 still scale well while phase 3 is slightly effected by increasing the number of cores, and the time-cost of phase 4 becomes extremely small. This is reasonable: (1) in *phase 1 & phase 2*, the operations are relying on the cardinalities of R and S at each node, but not the skew. (2) in *phase 3*, tuples are evenly distributed, which leads to the number of received query keys at each node not obviously changing when increasing the number

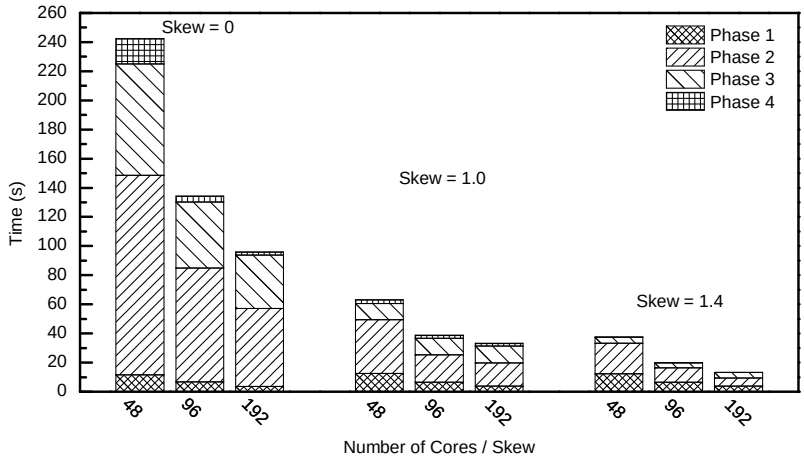


Fig. 6.8 The detailed time cost of query-based join approach on different key distributions by increasing number of cores.

of cores. Take the tuples with the same key k_1 for example, the $h_1(k_1)$-th node will always receive one k_1 from each node. It means that this node first receives 48 k_1 and then 96 k_1 when increasing the number of cores to 96. In the meantime, this increase will be leveraged by the decrease of the non-skewed query keys received at this node. (3) in *phase 4*, the size of the hash tables at each place built for S will decrease with the increment of the cores and the skew. That is why the time is only in the order of tens of *ms* when the skew is 1.4.

6.6 Evaluation of Outer Joins

In this section, we report the experimental results for outer joins following the same metrics as presented above. The platform information and the setup configuration of our evaluation are the same as the inner joins. We made some modifications for the test datasets: (1) we set the cardinality of R is set to 64M tuples, and (2) for the two skewed datasets of S, we vary the selectivity factors of the joins and set to 100% as the default value.

6.6.1 Runtime

We examined the runtime of four algorithms: the basic hash-based algorithm (referred as Hash), PRPD+Dup, PRPD+DER and our QC approach. We implement these tests using 16 nodes (192 cores) of the cluster on the default datasets with different skew.

Performance

The results in Figure 6.9 show that: (1) when S is uniform, the first three algorithms perform nearly the same and much better than our QC implementation; (2) with low skew, PRPD+DER becomes the fastest and our approach is better than the other two methods; and (3) with high skew, our approach outperforms the other three. In this process, the method PRPD+DER performs very well under skew, which confirms our expectation in Chapter 2. At the same time, the PRPD+Dup implementation shows the worst poor performance under skew, even worse than Hash, which means that skew handling techniques designed for inner joins can not always be applied for outer joins directly.

We also observe that with increasing of data skew, the time cost of Hash increases sharply while our scheme decreases sharply, which indicates that our QC approach has opposite properties compared with the commonly used hash-based join algorithm. In the meantime, although both the PRPD+Dup and PRPD+DER algorithms can be considered as hybrid methods on the basis of the conventional *hash-based* and *duplication-based* methods, the runtime of PRPD+Dup increases even more sharply than Hash, while PRPD+DER decreases with skew and shows its robustness against skew. This confirms that state-of-the-art optimization for outer joins can bring in significant performance improvements. QC performs the best under high skew conditions, where conventional methods fail. As such, our method can be considered as a supplement for the existing schemes. In fact, the optimizer in a system could pick the correct implementation based on the skew of the input so as to minimize runtime.

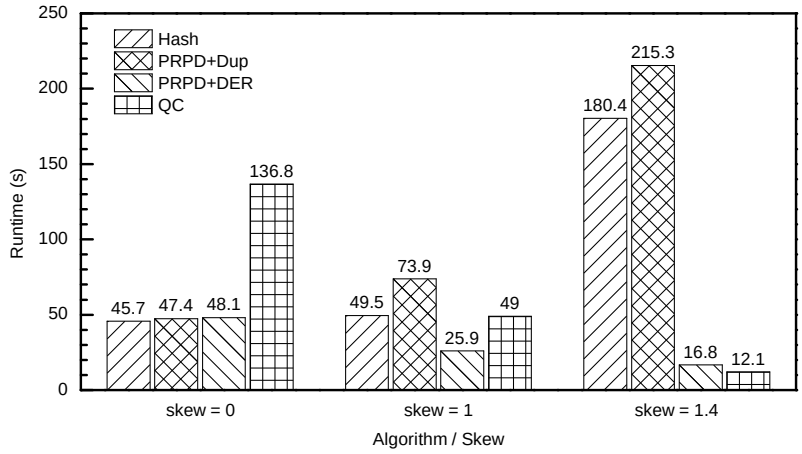


Fig. 6.9 Runtime comparison of the four algorithms under different skews (with selectivity factor 100% over 192 cores).

Selectivity Experiments

We also examine how join selectivity affects the performance for each algorithm. For both the low skew and high skew distributions, we created two different S that have the same cardinality as the default dataset but only 50% and 0% of the tuples join with a tuple in R.

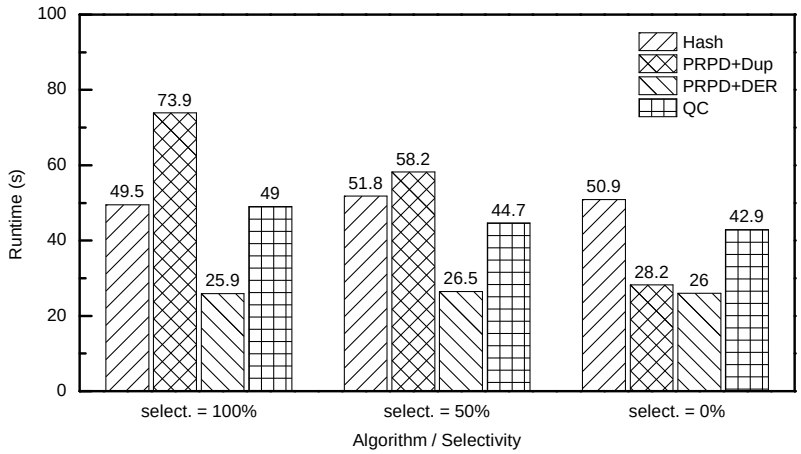


Fig. 6.10 Runtime of the four algorithms under low skew by varying the join selectivity factor ($skew = 1$ over 192 cores).

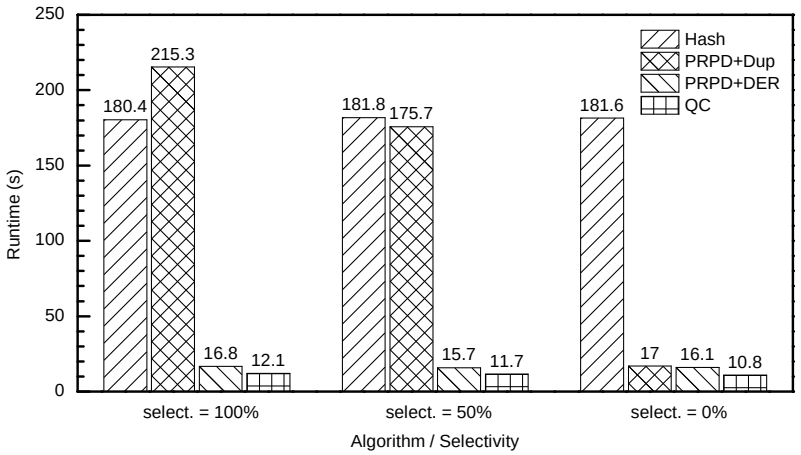


Fig. 6.11 Runtime of the four algorithms under high skew by varying the join selectivity factor ($skew = 1.4$ over 192 cores).

The results for the low skew dataset are presented in Figure 6.10. There, the PRPD+Dup algorithm shows lower runtime with decreasing selectivity, and the runtime of the other three methods does not change or slightly decreases. This is reasonable: (1) PRPD+Dup has to process the intermediate matched join results, the number of which depends on the join selectivity; (2) the transfer and join operations in Hash remain the same with different

selectivity; (3) though the number of the non-matched results increases with decreasing selectivity, PRPD+DER only needs to redistribute the non-matching row-ids for $R_{dup} \rtimes S_{loc}$, which remains small because R_{dup} is always small; and (4) the number of operations on *counters* and the final *result lookups* decreases with decreasing selectivity, leading to slightly performance improvement in our QC algorithm. These also appear when the dataset is highly skewed as shown in Figure 6.11. There, PRPD+Dup changes sharply, showing its sensitivity to the join selectivity. In contrast to this, our QC algorithm is robust and also outperforms the other three methods, demonstrating its strong ability in handling high skew in outer joins again.

6.6.2 Network Communication

Performance regarding communication costs is evaluated by measuring the number of received tuples. We implement our test on 192 cores, and collect the received tuples (keys) at each place by inserting counters. The results of the average number of received tuples for each place are shown in Figure 6.12.

We can see that all the four algorithms receive the same number of tuples when the dataset is uniform. This is reasonable, since all tuples in Hash, PRPD+Dup and PRPD+DER are processed only by redistribution as there is no skew and the number of query keys is equal to the number of total keys in our QC algorithm. With the increase in skew, the received tuples in Hash and PRPD+Dup does not change. In contrast, PRPD+DER and our method show a significant decrease, demonstrating they can handle the skew effectively. In addition, our method transfers much less data than PRPD+DER. All of this shows that our implementation can reduce the network communication more efficiently than other approaches under skew.

6.6.3 Load Balancing

With the same reason as we presented for inner joins, we analyze the load balancing of each algorithm based on the same metric, namely *number of received tuples (keys) at each place*. The responsible results are shown as Table 6.2. It can be seen that all four algorithms achieves perfect load balancing when the dataset is uniform. As the skew increases, the load balancing of the hash-based algorithm and PRPD+Dup becomes much worse. In the meantime, though PRPD+DER shows better improvement for that condition, our QC approach is still much better than PRPD+DER, which highlights the efficiency of skew handling again for the proposed query-based framework.

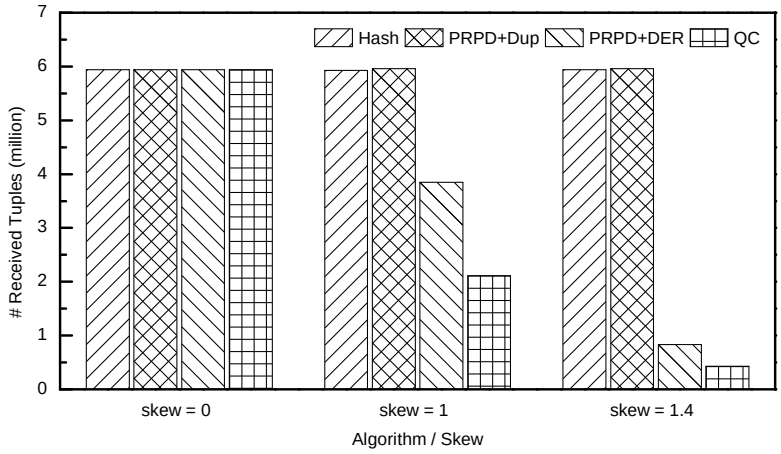


Fig. 6.12 The average number of received tuples (or keys) for each place under different skews (with selectivity factor 100% over 192 cores).

Table 6.2 The number of received tuples at each place (millions)

Algo.\skew	0		1		1.4	
	Max.	**Avg.**	**Max.**	**Avg.**	**Max.**	**Avg.**
Hash	5.94	5.94	62.40	5.93	347.78	5.94
PRPD+Dup	5.94	5.94	62.43	5.96	347.80	5.96
PRPD+DER	5.94	5.94	3.95	3.85	0.92	0.84
QC	5.94	5.94	2.12	2.12	0.43	0.43

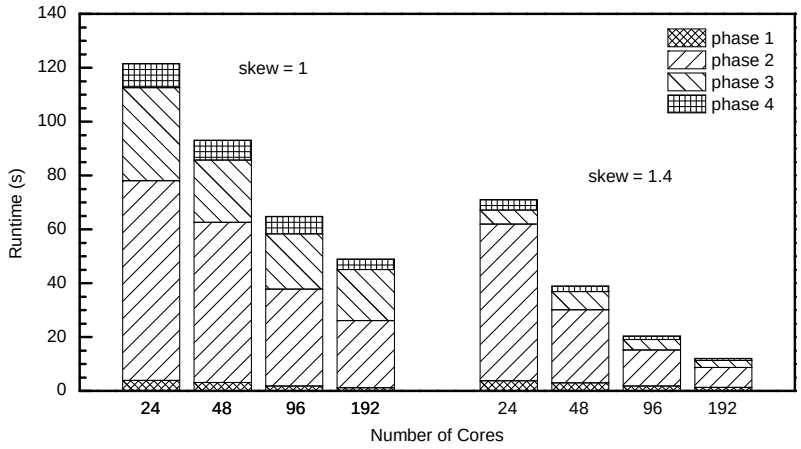


Fig. 6.13 The runtime breakdown of the QC algorithm under skews by varying number of cores (with selectivity factor 100%).

6.6.4 Scalability

We test the scalability of our QC implementation by varying number of processing cores under skew, from 24 cores (2 nodes) up to 192. The results are shown in Figure 6.13. It can be seen that the implementation generally scales well with the number of cores. Doubling number of cores brings in 1.30x - 1.44x speedup for the low skew dataset and 1.68x - 1.92x for high skew. In detail, phases 1, 2 and 4 scale well and phase 3 is slightly affected by increasing number of cores. The reason would be the same as what we have stated for the inner joins, namely the received unique keys at each place would also slightly increase with cores.

6.7 Conclusions

In this chapter, we have introduced a new framework for parallel joins, the *query-based distributed join*, which specifically targets joins with very high skew over distributed systems.

We have presented a detailed implementation of our approach using the X10 system. From these results, our main conclusions are that the proposed framework is: (a) *robust against data skew*, showing excellent load balancing, (b) *scalable*, speedup achieved with increments in the number of nodes (threads), (c) *highly efficient*, since we can process the join $256M \bowtie 1B$ with high skew in only 13 seconds, which is magnitudes faster compared with the conventional hash-based implementation, and also outperforms the state-of-art PRPD algorithm, and (d) *novel*, can be considered as a new approach and alternative to the two conventional frameworks commonly used.

Moreover, we have also extended the framework for processing large-large table outer joins and introduced a new outer joins algorithm, *query with counters*. The experimental results also show that our implementation is scalable and performs faster than the state-of-art PRPD+DER techniques [127] [126] under high skew.

As we see from the results that when the input is low skew or uniformly distributed, the query-based implementations becomes noncompetitive compared with the existing methods. To address this issue, in the next chapter, we will combine our method with approaches that partition data according to key skew, such as PRPD, so as to achieve more robust and even higher performance in the presence of different data skews.

Chapter 7

High Performance Skew-Resistant Parallel Joins in Shared-Nothing Systems

7.1 Introduction

In the previous chapter, from the analysis as well as the experimental results we can see that the state-of-art join methods designed to handle data skew over distributed systems offer significant improvements over naive implementations, and the join performance could be further improved using the proposed query-based joins in the presence of high skew. However, our method encouters a performance bottleneck when processing low-skewed datasets. The reason is that the number of transferred keys and retrieved values will be extremely large when processing large-scale data, and the two-sided communication decreases the performance consequently.

To achieve an efficient and robust join performance over a range of skew conditions on distributed systems, in this chapter, we further refine the query-based approach and present a new join algorithm called PRPQ (*partial redistribution & partial query*). Similar to the *query with counter* algorithm described in the previous chapter, the PRPQ method can also be easily applied to outer joins. Regardless, we are more interested in the performance and characteristic of each join framework (or distributed join patterns), in terms of computation and network communication during join executions. Therefore, compared to the last chapter, we only focus on studying inner joins.

In this work, we also conduct a performance comparison of four parallel join algorithms (the three methods evaluated previously and the new proposed PRPQ) based on a theoretical analysis of their implementations. Moreover, we present more exact and more detailed experiments for our method over different large datasets. The experimental results

demonstrate that the proposed PRPQ algorithm is indeed robust and scalable under different skew conditions. Specifically, compared to the state-of-art PRPD [127] method, it achieves $16\% - 167\%$ performance improvement with $24\% - 54\%$ less network communication under different join workloads, figures that confirm the theoretical analysis.

The rest of this chapter is organized as follows: In Section 7.2, we introduce our PRPQ algorithm. We conduct a theoretical analysis of different parallel join approaches in Section 7.3. The detailed implementation of our algorithm is presented in Section 7.4. Section 7.5 provides a quantitative evaluation of our approach while Section 7.6 concludes the chapter.

7.2 PRPQ Joins

In this section, we first present the PRPQ algorithm, and then conduct an intuitive comparison with the *query-based* method and the state-of-art PRPD algorithm.

7.2.1 The PRPQ Algorithm

The PRPQ joins can be considered as a hybrid approach based on both the hash-based and query-based implementation. With the same assumption as previously that the input relation R is uniformly distributed and S is skew. In PRPQ, the relation R is distributed in a similar fashion as in the query-based algorithm, namely, all the tuples are redistributed to the responsible nodes according to the hash value of their join keys. However, the relation S at each node is split into two parts according to its skewed keys: (1) the high skewed part h (with tuples whose keys appear more times than the specified threshold) is processed with the query-based scheme, and (2) the remaining part S', that is processed by the hash-based implementation.

As shown in Figure 7.1, the tuples in each R_i, h_i and S'_i at each node i are first hash-partitioned based on the hash value of their join keys. After the redistribution of R and the building of the hash table R_k at each node, the keys in each h_{ik} will query all remote R_k to retrieve their responsible values and formulate the responsible outputs. On the other hand, the S'_{ik} will be distributed to the k-th node and join with R_k there. Note that, as h_{ik} are all entirely independent from each other and only query their corresponding remote nodes (which is the same as the query-based implementation), we do not need any global operation, instead, we only need to quantify the **local** skew at each node, which can be easily achieved.

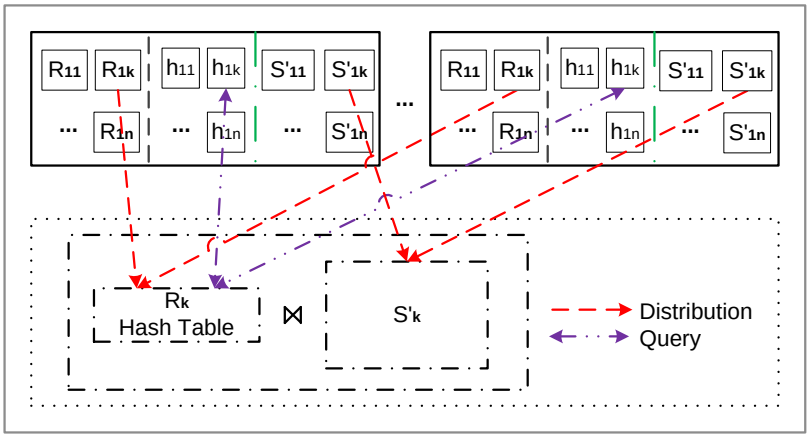


Fig. 7.1 The PRPQ join approach. Only the high skew part of S implements the query operations, and the rest is processed as the basic hash method.

7.2.2 Compared to the QUERY-BASED Algorithm

We apply the query scheme only for the high skew tuples and all the low skew tuples are just simply redistributed. Even when there exists large numbers of low skewed tuples, the number of query keys and returned values will be still small. Therefore the PRPQ can efficiently remedy the shortcoming of the basic query algorithm and improve its robustness.

Moreover, inheriting from the advantages of the basis query algorithm, PRPQ can also highly reduce the network communication when processing skewed data. The reason is that none of the highly skewed tuples are distributed, but only their unique keys as well as the corresponding returned values, which are always very small. Furthermore, as PRPQ adopts the complementary advantages of both hash-based and query-based implementations, the method should, for any kind of inputs, outperform both algorithms for a suitable threshold t. We will exam this conjecture through our evaluation in Section 7.5. In addition to this, PRPQ has an extra operation, namely quantifying the skew to partitioning the tuples. However, we only need to quantify the **local** skew (namely for each S_i) at each node, which can be easily achieved.

7.2.3 Comparison with PRPD

Similar to the query-based approach, taking a higher level comparison with the PRPD [127] method as described, PRPQ also has two main advantages on (1) skew quantification and (2) redundancy removement, as presented in Chapter 6.

In fact, the first advantage means our PRPQ method will be more flexible or more efficient in the face of different join workloads, especially for the unevenly distributed ones.

Taking an extreme condition for example, for a 10^6-node system, if a key in S follows the linear distribution over the computation nodes (e.g. appearing 10^6 times on the first node, $10^6 - 1$ on the second node etc. and only 1 time on final node), then *how can we define the global skew using PRPD?* [127] proposes a solution that redistributes the skew tuples evenly to all the nodes before the join. However, this pre-redistribution will generate extra communication costs, while more complex and careful global statistical operations for all tuples of S are required. The authors in [127] do not provide any detailed implementation or experimental details regarding this pre-processing. Therefore, for PRPD in what follows, we do not consider any rebalancing operations for the uneven skew of S but just adopt a general method, namely each node just broadcasts its local skew keys so as to organize the global skew. In contrast, by simply using a threshold such as $10^6/2$, PRPQ will know that the key is skew in the first half million nodes and non-skew for the remaining nodes.

Moreover, in the condition where there are many mid-skewed tuples, for instance, the relation S_i at each node i contains 1 million totally different unique keys (assuming uniform distributed) where each key appears 40000 times, then, *should we consider these keys as skew?* If so, each node under the PRPD scheme has to broadcast the responsible 1 million tuples of R_i to all the nodes, which means that each node will receive $10^6 \cdot 10^6 = 10^{12}$ tuples over the 10^6-node system. In comparison, using PRPQ, each node just receives $(10^6/10^6) \cdot 10^6 = 10^6$ keys and the corresponding 10^6 values. This indicates that PRPQ can further significantly reduce the network communication (considering a key or value as a half tuple) and potentially improve the join performance over PRPD. We will demonstrate this detailed performance difference using different workloads in our evaluations in Section 7.5.

Additionally, the main difference between the PRPQ and PRPD algorithm is in processing skewed tuples, namely using *query*, a duplication-free way, to replace the conventional *duplication* method, thus the extension or theoretical analysis from PRPD [127] can be applied to our approach directly. For example, regarding the *skewed-skewed* joins, similar to the approach taken in PRPD, if R is skewed, the skewed part of R can be used to query the corresponding non-skewed part of S, the skewed part of S can be used to query the corresponding non-skewed part of R, and others would be hash-redistributed, for our PRPQ method.

7.3 Theoretical Comparison of Parallel Join Approaches

To support our above assertions, and also to conduct a more valuable comparison of different distributed join patterns, in this section, we conduct a theoretical performance analysis of

the four parallel join approaches: the most commonly used hash-based algorithm, the state-of-art PRPD algorithm, our basic query-based joins and the refined PRPQ algorithm. We track their computation and communication costs over a distributed architecture for a join implementation and give an insight into their characterization in the presence of skew data. Specifically, we also compare the performance of the PRPQ algorithm versus PRPD.

7.3.1 Skew in Parallel Joins

We begin by developing some analysis of skew and its impact on parallel joins by considering a common parallel DBMS (PDBMS) with $n \geq 2$ computing nodes (threads) over which we organize a simple data model. We assume tuples in R and S are simply $<key, payload>$ pairs and *key* is their join attribution. If there are many tuples in S that have the same key but with different payload, then S is considered as skew data. For the purpose of the following analysis we impose that relation R is uniformly distributed while S is skewed and let the number of keys in each S_i ($i \in [1,n]$) follow the distribution function $f(r)$, where r is the rank of a key according to its frequency of occurrence and $f(r) > f(r+1)$.

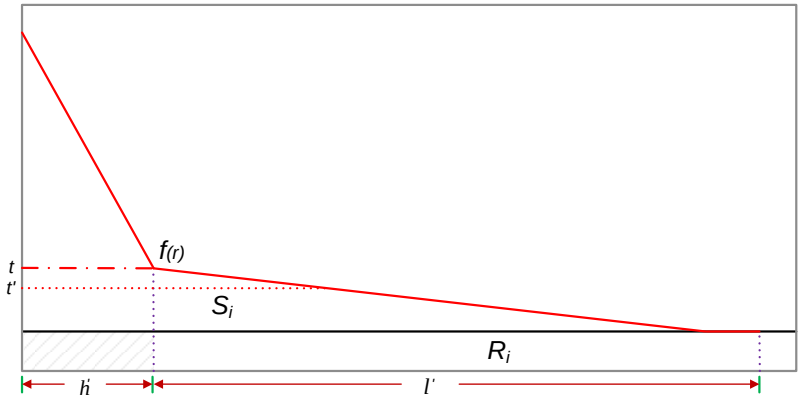


Fig. 7.2 Distribution of the tuples in S at each node based on the rank of keys.

As shown in Figure 7.2, we divide the keys of S in two parts: the high skew part with h' keys and the low skew part with l' keys. To highlight the skew, we assume that $h' << l'$ and the high skew tuples form a large part of relation S. A typical example for this kind of input is one where the data follows the Zipf distribution with high skew.

In the hash-based framework, the redistribution of tuples in relations R and S deeply relies on the hash function, and all the tuples with the same join attribute will be transferred to the same remote node. To simplify the tracking of tuples after the redistribution, we define E_i^j as the key at the i-th node with rank j, and assume that $E_i^j \equiv j \pmod{n}$. With

the hash function $h(E) = E\%n$ for tuple transferring, tuples with the rank $(k + x \cdot n)$ will be flushed into the k-th node. On this basis, we set $h' = h \cdot n$ and $l' = l \cdot n$, then the number of tuples[1] N_k^1 the k-th node receives will be:

$$\begin{array}{lll} & (|R_k|/n) \cdot n & \text{distribution of R} \\ + & [\sum_{i=0}^{h+l-1} f(k+i \cdot n)] \cdot n & \text{distribution of S} \\ = & |R|/n + n \cdot \sum_{i=0}^{h+l-1} f(k+i \cdot n) & \end{array}$$

We note that the number of hash operations in join operations C_k^1 at the k-th node would be equal to N_k^1, since the total number of hash operations for $A \bowtie B$ is $|A| + |B|$, where $|A|$ is for adding (hash table building) and $|B|$ is for reading (hash table probing).

The differences in the number of received tuples and hash operations between the k-th and $(k+1)$th-node is $\Delta_1 = n \cdot \sum_{i=0}^{h+l-1} [f(k+i \cdot n) - f(k+1+i \cdot n)] > 0$, which means that the 1-st node receives the highest number of tuples with the highest number of hash operations, while the n-th node receives the lowest respectively. This leads to the scenario where the first few nodes are prone to becoming hot spots resulting in performance bottlenecks. The performance hit arise due to: (1) communication costs as large numbers of tuples are transferred to hot spots over the network, and (2) load imbalance: a large number of hash table lookups are implemented at hot spots in the probing phase. Such issues impact system scalability which will be reduced as employing new nodes cannot yield improvements because large number of skew tuples will remain distributed to the same nodes. For example, with increasing n, the large number of tuples with key E_i^1 will still flood into the first node in this case.

7.3.2 PRPD Joins

The detailed analysis and implementation of the PRPD algorithm has been presented in Chapter 2. Using the data model in Figure 7.2, the relation S is split into two parts based on the threshold t set for the frequency of occurrence of a key: (1) a locally kept part S_{loc}; tuples with $f(r) \geq t$ are considered as *high skew* and are kept locally, not participating in the redistribution phase, and (2) the redistributed part S_{dis}; the remaining tuples with $f(r) < t$ are redistributed as in the basic hash-based implementation. In the meantime, the relation

[1] In our analysis, we denote the algorithms under consideration with a superscript in the cost of N_k and C_k and with a subscript for Δ. We use the numerals 1, 2, 3, 4 for this purpose. For example, N_k^1 refers to N_k for the hash algorithm, C_k^2 refers to C_k for the PRPD algorithm etc.

R is divided into two parts as well: (1) the duplicated part R_{dup} (shown in Figure 7.2 as the shaded area); the tuples which contain the keys in S_{loc} are broadcast to all the nodes, and (2) the redistributed part R_{dis}; the remaining part of R which is redistributed as normal. After the duplication and the redistribution operations, the final join is composed by $R_{dis} \bowtie S_{dis}$ and $R_{dup} \bowtie S_{loc}$ respectively at each node.

As the split of R and S fully relies on the skew keys in S and the final join will fail if any node does not have global knowledge of such keys, broadcasts for the skew keys at each node are required initially. Based on this, each node in PRPD will receive $hn \cdot n$ keys in the skew sharing process. It is possible that a portion of such received keys are repetitive. To quantify this, we assume that the repetition appears only on keys with the same rank and the ratio $\beta = |\cup_{i=1}^{n} E_i^j|/n$ is the same for each rank, then the number of received *unique* keys will be $\beta \cdot hn^2$ at each node. In fact, β indicates whether the skew tuples are evenly partitioned. Noting that β is in the range $[1/n, 1]$, we distinguish two extreme cases: (1) $\beta = 1/n$: $E_i^j = E_{i'}^j$ for $\forall i, i' \in [1,n]$, thus all skew tuples are replicated evenly over each node; (2) $\beta = 1$: skew keys are unique to each node.

To simplify the split of R_i in PRPD, we make an additional assumption that the received βhn^2 unique keys are uniformly distributed. Therefore, the duplication part of R_i is $\beta hn^2/n$ at each node and tuples in the remaining part R_{i_dis} will be uniformly distributed as well. We consider a single key or value as $1/2$ tuple in terms of its size, then the number of received tuples N_k^2 at the k-th node is:

$$
\begin{array}{lll}
 & hn \cdot n/2 & \text{duplication of skew keys} \\
+ & \beta hn \cdot n & \text{duplication of } R_{dup} \\
+ & [(|R_k| - \beta hn)/n] \cdot n & \text{distribution of } R_{dis} \\
+ & [\sum_{i=h}^{h+l-1} f(k + i \cdot n)] \cdot n & \text{distribution of } S_{dis} \\
= & |R|/n + hn(n/2 + \beta n - \beta) + n \cdot \sum_{i=h}^{h+l-1} f(k + i \cdot n) &
\end{array}
$$

As both R_i and S_i are split based on received hn^2 skew keys, we treat this process the same as a join (i.e. as a lookup S_i and R_i over a `hashset` composed by the received keys), and take into account the number of hash operations. Consequently, the number of hash operations C_k^2 for the k-th node can be written as:

$$
\begin{array}{lll}
 & |S_k| + hn^2 & \text{split of S} \\
+ & |R_k| & \text{split of R} \\
+ & (|R|/n - \beta hn) + n \cdot \sum_{i=h}^{h+l-1} f(k+i\cdot n) & R_{dis} \bowtie S_{dis} \\
+ & \beta hn^2 + \sum_{i=1}^{hn} f(i) & R_{dup} \bowtie S_{loc} \\
= & \mathbb{C} + |R|/n + hn \cdot (n + \beta n - \beta) &
\end{array}
$$

where $\mathbb{C} = |R|/n + |S|/n + \sum_{i=1}^{hn} f(i) + n \cdot \sum_{i=h}^{h+l-1} f(k+i\cdot n)$.

The differences in received tuples and hash operations between the k-th and $(k+1)$-th node is the same, i.e. both are $\Delta_2 = n \cdot \sum_{i=h}^{h+l-1} [f(k+i\cdot n) - f(k+1+i\cdot n)]$. Recall that the subscript 2 here refers to the PRPD algorithm. Clearly, $\Delta_2 > 0$, suggesting that the load imbalance still exists in PRPD. However, compared to the basic hash algorithm: $\Delta_1 - \Delta_2 = n \cdot \sum_{i=0}^{h-1} f(k+i\cdot n) > 0$. In other words, the load imbalance between each node in PRPD is smaller than that in the hash-based approach. In fact, Δ_2 indicates that the load imbalance is the result of and occurs in the low skew part, where the difference between each $f(r)$ is always very small, and as a consequence, PRPD will always achieve good load balancing. When the low skew part is uniformly distributed, nodes in PRPD will be fully load balanced.

7.3.3 Query-based Joins

Following the detailed work flow and implementation of the query-based joins as presented in Chapter 6, the number of tuples that the k-th node receives N_k^3 is:

$$
\begin{array}{lll}
 & (|R_k|/n) \cdot n & \text{distribution of R} \\
+ & [(hn+ln)/n] \cdot n \cdot 1/2 & \text{query of skew keys} \\
+ & [(hn+ln)/n] \cdot n \cdot 1/2 & \text{returned values} \\
= & |R|/n + (h+l) \cdot n &
\end{array}
$$

We can see that N_k^3 here is a constant and totally independent from the node location k, implying that each node is always fully load balanced in terms of communications. Correspondingly, if we do not consider the `iteration` operation on hash tables to extract the keys, then the number of hash operations C_k^3 for the k-th node is:

$$
\begin{array}{lll}
 & |R|/n+(h+l)\cdot n & \texttt{join} \text{ between R and query keys} \\
+ & (h+l)\cdot n+|S_i| & \texttt{join} \text{ between returned values and S} \\
= & |R|/n+|S|/n+2(h+l)\cdot n &
\end{array}
$$

This shows that C_k^3 is also a constant, and that each node is fully load balanced. Therefore, there are no hot spots in our query-based framework. In the meantime, we also notice that $(h+l)\cdot n$ intermediate elements in the basic query algorithm take part in the `join` twice, both at the local and remote nodes, a number that could yield a heavy workload. The difference between the amount of computation required by the algorithm and that required by the hash-based algorithm is $\Delta_{31} = C_k^3 - C_k^1 = |S|/n + n\cdot[2(h+l) - \sum_{i=0}^{h+l-1} f(k+i\cdot n)]$. When S_i is low skewed, namely when $f(r) \approx f(r+1) = c$, then $\Delta_{31} = 2(h+l)\cdot n > 0$. In this case, the number of unique keys at each node will be $(h+l)\cdot n = |S_i|/c$. Obviously, for a fixed size input, when c is very small such as $c = 1$, the $\Delta_{31} = |S_i|$ will be much greater than 0, implying a much heavier computational load of query-based joins. Furthermore, though there exists $N_k^3 = N_k^1$ in this scenario, the two-way communication of a large number of $|S|/n$ keys (or values) could also be more costly than the one way transmission of $|S|/n$ tuples. This suggests that when the input is low skew with low repetitive keys, such as when uniformly distributed, the query-based joins will have much heavier work load than the hash-based joins and could also spend more time on communication. This also clearly clarifies why query-based joins could meet performance bottlenecks in the the face of non-skew data.

7.3.4 PRPQ Joins

Based on the workflow of PRPQ, with the knowledge of local skew, the number of received tuples N_k^4 at the k-th node is:

$$
\begin{array}{lll}
 & (|R_k|/n)\cdot n & \text{distribution of R} \\
+ & [\sum_{i=h}^{h+l-1} f(k+i\cdot n)]\cdot n & \text{distribution of } S_{dis} \\
+ & (hn/n)\cdot n\cdot 1/2 & \text{query of skew keys} \\
+ & (hn/n)\cdot n\cdot 1/2 & \text{returned values} \\
= & |R|/n+hn+n\cdot\sum_{i=h}^{h+l-1} f(k+i\cdot n) &
\end{array}
$$

For computation, we also need to split S over the local high skew keys, and the hash table R_k only needs to be built once so that both supports the query and join operations. Using the previously defined $\mathbb{C}$, the relative number of C_k^4 is:

$$\begin{array}{lll} & |S_k| + hn & \text{split of S} \\ + & |R|/n + n \cdot \sum_{i=h}^{h+l-1} f(k+i \cdot n) & R_{dis} \bowtie S_{dis} \\ + & hn & \text{lookup query keys} \\ + & hn + \sum_{i=1}^{hn} f(i) & \texttt{join} \text{ of returned values and S} \\ = & \mathbb{C} + 3hn & \end{array}$$

We note that the differences in communication and computation between each node in PRPQ depends only on the low skew tuples, an observation which also explains the load-balancing performance of PRPD. Moreover, we apply the query scheme only for the high skew tuples and all the low skew tuples are just simply redistributed. Even when there exists large numbers of low skewed tuples, we can see that the number of query keys and returned values is hn. This is still relatively small and efficiently remedies the shortcoming of the basic query algorithm and improves its robustness. As PRPQ adopts the complementary advantages of both hash-based and query-based implementations, the method should, for any kind of inputs, outperform both algorithms for a suitable threshold t. We will exam this conjecture through the evaluation given in Section 7.5.

7.3.5 Performance Comparison

We focus on comparing the performance of our PRPQ algorithm with the state-of-art PRPD. We assume that both algorithms have the same threshold configuration, and each node has a priori knowledge of its local skew keys hn.

With respect to network communication, the load balancing of PRPD and PRPQ is similar as they both depend on the low skew part. Additionally, decreasing the threshold t (t' shown in Figure 7.2) can further improve load balancing, because the low skew part decreases, leading to a smaller difference between each node - Δ_2 and Δ_4 in both algorithms. By comparing the network load, the difference of received tuples between PRPD and PRPQ at each node is:

$$\Delta N_{24} = N_{prpd} - N_{prpq} = [(n-1) \cdot \beta + (n/2-1)] \cdot hn \geq 0$$

In other words, PRPQ always has less network communication than PDPR, while $N_{prpd} = N_{prpq}$ only when $h = 0$ (i.e. no skew tuples are detected) and therefore all tuples are processed by the basic hash algorithm.

Moreover, by increasing hn (namely decreasing t), both algorithms can achieve better load balancing, however, N_k^2 and N_k^4 will increase, suggesting that there is a trade-off between communications and load-balancing. On the other hand, the cost of the increase in communication is $O(n)$ for PRPD while $O(1)$ for PRPQ, showing the network load associated with PRPD is more sensitive to load balancing than PRPQ. In contrast to PRPD, PRPQ is agnostic of the factor β, which demonstrates that the network load of this scheme is more robust to dataset partitioning (no matter how evenly the skew tuples are partitioned).

Similarly, for computation, PRPD and PRPQ achieve absolute balancing only when the low skew keys are uniformly distributed. Since normally the number of high skew keys in S_i will be relatively small compared to $|R_i|$, namely, $|R_i| > hn$, the difference of the hash operations between PRPD and PRPQ for each node is:

$$\begin{aligned}\Delta C_{24} &= C_{prpd} - C_{prpq} \\ &= (|R|/n - hn) + [(n-1)\cdot\beta + (n-2)]\cdot hn \geq 0\end{aligned}$$

This demonstrates that PRPQ has less computation overhead than PDPR under all conditions. Additionally, as in the analysis of network communication, $C_{prpd} = C_{prpq}$ only when $h = 0$, while increasing h, β or n can amplify their difference.

We assume the time cost $t(x)$ to transfer x tuples is $t(x) = \delta_0 + \delta_1 \cdot x$, where δ_0 is a constant that represents the latency for each data transfer while δ_1 is the time for transferring a single tuple. Because of the *all-to-all* communication in PRPD (broadcast of keys and tuples) and PRPQ (push keys and return values) both take place twice, here we just need consider their time difference on transferring the data, namely $\delta_1 \cdot x$. Combining with the above two equations, if we define the time cost of an **ideal** single hash table operation as λ_1, then we can calculate the time difference between PRPD and PRPQ:

$$\begin{aligned}\Delta T_{24} &= \delta_1 \cdot \Delta N_{24} + \lambda_1 \cdot \Delta C_{24} \\ &= \delta_1 \cdot [(n-1)\cdot\beta + (n/2-1)]\cdot hn + \\ &\quad \lambda_1 \cdot [(n-1)\cdot\beta + (n-3)]\cdot h + \lambda_1 |R|/n\end{aligned}$$

As $N_{23} \geq 0$ and $C_{23} \geq 0$, clearly $\Delta T_{23} \geq 0$. In other words, PRPQ can outperform PRPD through efficiently reducing both the network communication and computation in joins, rendering PRPQ more suitable for modern high performance computing systems. Moreover,

in the general case where $n \geq 3$, ΔT_{23} increases with increasing n, β or h. For example, the time difference between PRPD and PRPQ is 3 seconds when n=100, if you increase n to 200, the time difference could become 6 seconds. This implies that PRPQ has more robust performance under different conditions. For instance, in scenarios that involved the processing of massive uncertain datasets over a large computing center, both the number of high skew keys and the nodes will be large, and the skew tuples could be heavily unevenly partitioned as well. This would render our proposed PRPQ method a better choice than the PRPD algorithm under such a scenario.

7.4 Implementation

We present the detailed implementation of the PRPQ algorithm using the X10 framework. As extracting skew tuples at each node is based on local skew quantification, we add in the parameter *threshold* in our implementations, namely the number of occurrences of a key after which the corresponding tuples are considered as skew tuples. We first discuss how we deal with this parameter and then describe the PRPQ implementation.

7.4.1 Local Skew

There are various ways to measure local skew quickly, such as sampling, scanning etc. However efficient skew measurement does not concern us here and so we just count key occurrences and store them in descending order at each node in a flat file. In each test with parameter t, each node will pre-read the responsible keys (keys appear more than t times) in an `ArrayList` and consider them as the required skew keys. These pre-processes make the performance comparison more fair and meaningful because: (1) The total join performance is very sensitive to the chosen skew keys and operations like sampling cannot guarantee the same set of keys are selected, and (2) the extra time cost for skew extraction is removed, so that the focus is on analyzing runtime performance only.

7.4.2 Parallel Processing

Similar to the query-based joins, the implementation of PRPQ is divided into the following four phases as well.

R Distribution: We are interested in high performance distributed memory join algorithms, therefore, we first read all the tuples in an `ArrayList` at each node, and then commence

Algorithm 14 R Distribution

```
1: finish async at p ∈ P {
2: Initialize R_c:array[array[tuple]](n)
3: for tuple ∈ list_of_R do
4:     des=hash(tuple.key)
5:     R_c(des).add(tuple)
6: end for
7: for i ← 0..(n − 1) do
8:     Push R_c(i) to r_R_c(i)(here) at place i
9: end for
10: }
```

Algorithm 15 Push Query Keys

```
1: finish async at p ∈ P {
2: Initialize T:array[hashmap[key,ArrayList(value)]](n)
           S'_c:array[array[tuple]](n) and skew:hashset[key]()
3: Read the skew keys in skew based on t
4: for tuple ∈ list_of_S do
5:     des=hash(tuple.key)
6:     if tuple.key ∈ skew then
7:         Add tuple in T(des)
8:     else
9:         Add tuple in S'_c(des)
10:    end if
11: end for
12: for i ← 0..(n − 1) do
13:    Extract keys in T(i) to key_c(here)(i)
14:    Push key_c(here)(i) to remote_key(i)(here),
           S'_c(i) to r_S'_c(i)(here) at the place i
15: end for
16: }
```

distribution of the relation R. The pseudocode of this process is given in Algorithm 14. The array R_c is used to collect the grouped tuples, and its size is initialized to the number of computing nodes n. Then, each thread reads the `ArrayList` of R and groups the tuples according to the hash values of their keys. Next, the grouped items are sent to the corresponding remote place.

Push Query Keys: The implementation of the second step is given in Algorithm 15. The skew keys are first read into a `hashset` based on the parameter t. Next all the tuples in S will be checked for skew such that `hashmap` collects the skew tuples while the arrays S'_c collects the non-skew tuples. After processing all the tuples, the keys in each hash table will

Algorithm 16 Return Queried Values

```
1: finish async at p ∈ P {
2: Initialize T':hashmap, value_c:array[value]
3: for i ← 0..(n−1) do
4:     Put received tuples of r_R_c(here)(i) into T'
5: end for
6: for i ← 0..(n−1) do
7:     Lookup received r_S'_c(i) in T'
8:     Output join results of non-skew part
9: end for
10: for i ← 0..(n−1) do
11:     for key ∈ remote_key_c(here)(i) do
12:         if key ∈ T' then
13:             value_c.add(T'.get(key).value)
14:         else
15:             value_c.add(null)
16:         end if
17:     end for
18:     Push value_c(i) to r_value_c(i)(here) at place i
19: end for
20: }
```

Algorithm 17 Result Lookup

```
1: finish async at p ∈ P {
2: for i ← 0..(n−1) do
3:     for value ∈ r_value_c(here)(i) do
4:         if value ≠ null then
5:             Look corresponding key in T(i)
6:             Output join results of the skew part
7:         end if
8:     end for
9: end for
10: }
```

be extracted by an iteration on its `keyset`. These keys will be kept in *key_c*, the same as S'_c, both are pushed to the assigned place for further processing.

Return Queried Values: The implementation of this phase at each place is similar to a sequential hash join. The received tuples and key arrays, representing the distributed R, S' and grouped query keys respectively. For the tuples, all the $<key,value>$ pairs of R are placed in the local hash table T', and S' looks up the match in T' to output the join results for the non-skew tuples. Meanwhile, the query keys access T' sequentially to get their values.

In this process, if the mapping of a key already exists, its value is retrieved, otherwise, the value is considered as *null*. In both cases, the value of the query key is added into a temporary array so that it can be sent back to the requester(s). The details of the algorithm are given in Algorithm 16.

Result Lookup: The join results for the skewed tuples can be looked up after all the values of the query keys have been pushed back. Since the query keys and their respective values are held in order inside arrays, we can easily look up the keys in the corresponding hash tables to organize the join results as shown in Algorithm 17. The entire join process terminates when all individual activities terminate.

7.5 Experimental Evaluation

Platform. Our experiments were executed on the *High Performance Systems Research Cluster* in IBM Research Ireland. Each computation unit of this cluster is an iDataPlex node with two 6-core Intel Xeon X5679 processors running at 2.93 GHz, resulting in a total of 12 cores per physical node. Each node has 128GB of RAM and a single 1TB SATA hard-drive and nodes are connected by a Gigabit Ethernet. The operating system is Linux kernel version 2.6.32-220 and the software stack consists of X10 version 2.3 compiling to C++ and gcc version 4.4.6.

Datasets. The datasets used as benchmarks were chosen to mimic joins in decision support environments. We mainly focus on the most expensive operation in such scenarios: the join between the intermediate relation R (the outcome of various operations on the dimension relations) with a much larger fact relation S [16]. We fix the default cardinality of R to 64M tuples and S to 1B tuples. Because data in warehouses is commonly stored following a column-oriented model, we set the data format to $<$*key, payload*$>$ pairs, where both the *key* and *payload* are 8-byte integers.

As the primary keys in R should be unique, we only add skew to the corresponding foreign keys in S. We list the input of S in the table below in bold font indicating default values. For the Zipf distribution, the skew factor is set to 0 for uniform, 1 for low skew (top ten popular keys appear 14% of the time) and 1.4 for high skew (top ten popular keys appear 68% of the time). Such workloads are common in recently studies [16, 20, 75]. For the linear distribution case, the $f(r) = 46341 - r$ is for low skew while $f(r) = 23170$ is for uniform but highly repetitive; both contain 46341 unique keys (1B tuples). When S is uniform, the tuples are created such that each of them matches the tuples in the relation R

with the same probability. For the skew datasets, the **unique** keys of tuples are uniformly distributed and each of them has a match in R. Moreover, we distribute all the tuples in R evenly to all computing nodes while we use both even and sort-range[2] methods for S.

Table 7.1 Datasets with different key distribution and partitioning used in our tests

S	Key distr.	Partition	Size
Zipf	*skew* = **0**, **1**, **1.25**, **1.4**	**evenly**,	512M,
Linear	$f(r) = 46341 - r$, 23170	sort-range	**1B**, 2B

Setup. We set the system parameter `X10_NPLACES` to the number of cores and the `N_Thread` to 1, namely one place for one single activity, which avoids the overhead of context switching at runtime. In all experiments, we only count the number of matches, but do not actually output join results. Moreover, for PRPD and PRPQ, we implemented a test series with different t for each data set, as shown in Figure 7.4. When we present the results, we always choose the point t with the best achieved run time.

7.5.1 Runtime

We consider the runtime of the four algorithms: the hash-based algorithm (referred as *Hash*), PRPD [127], PRPQ and the basic query approach (referred as *Query*). We implement these tests using 16 nodes (192 hardware cores) of the cluster on the default datasets.

Performance

The results in Figure 7.3 illustrate that: (1) when S is uniform, the *Hash*, PRPD and PRPQ algorithms perform nearly the same and much better than the *Query* implementation, which matches our analysis about the shortcomings of the query-based joins; (2) with low skew, PRPD and PRPQ is comparatively faster that the other two approaches; and (3) with high skew, *Hash* is the worst while the other three perform much better, demonstrating their capacity to handle skew.

It can also be seen that with increasing data skew, the time cost of *Hash* increases sharply while that of *Query* decreases. This demonstrates that *Query* is more suitable for processing high skew datasets. Moreover, PRPD and the PRPQ algorithm change much more smoothly compared to the two basic approaches and their time cost decreases with increasing skew,

[2]All tuples are sorted according to the rank of the keys, and then equal-sized partitioning based on the number of places.

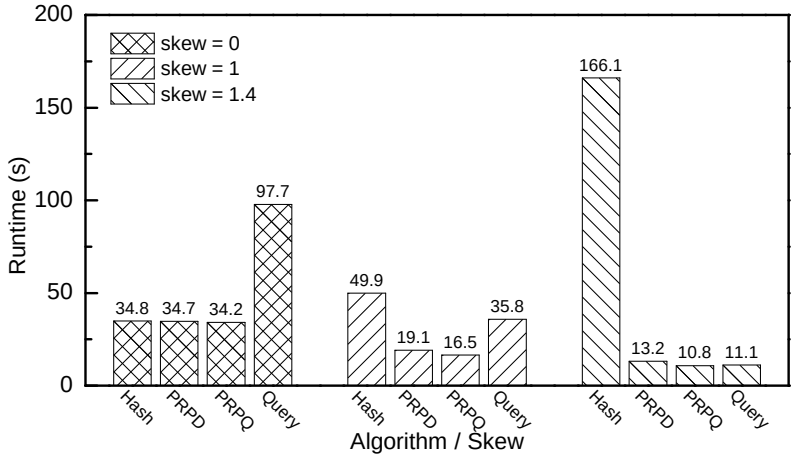


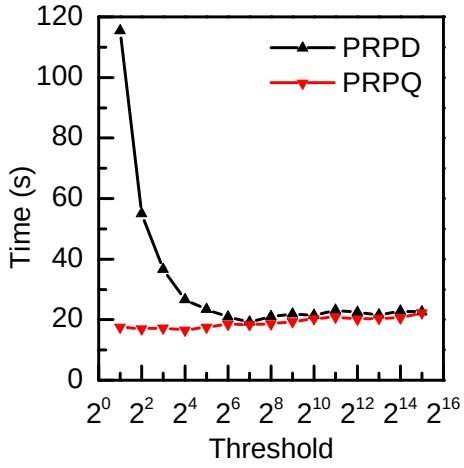
Fig. 7.3 Runtime of the four algorithms.

demonstrating their robustness against skew. Furthermore, for the high skew case, we note that PRPQ outperforms *Query* with threshold $t = 32$, which implies that those tuples with keys appearing less than 32 times perform better in *Hash* than in *Query*. This confirms our analysis and expectation described in Section 7.2.2.

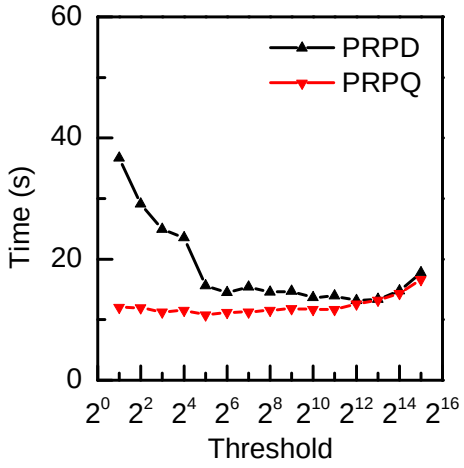
PRPQ vs PRPD

Figure 7.3 also shows that the best performance achieved by PRPQ is better than PRPD under different skew scenarios. To conduct a more detailed comparison, we implemented a series of tests on different datasets and with different partitioning strategies. The threshold t ranges between values that enable us to always capture the skew keys and present the results in Figure 7.4 where: (1) *evenly* refers to S being evenly distributed to all the nodes; (2) *range* refers to the sort-range partitioning; (3) *Linear 0* means that S follows the linear distribution $f(r) = 23170$ while *Linear 1* refers to $f(r) = 46341 - r$; (4) the first two numbers in brackets indicate the value of t for which the best performance is achieved by PRPD and PRPQ respectively while the third one demonstrates the relative speedups of PRPQ over PRPD based on their best runtime.

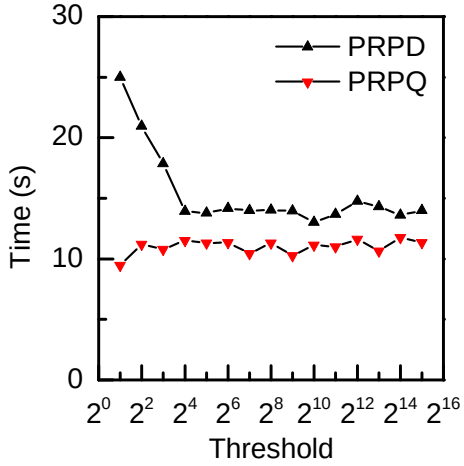
We can see that, for any given t, PRPQ always performs better than PRPD. This is consistent with our theoretical analysis in Section 7.3.5, and highlights again the fact that PRPQ will always be faster than PRPD. Looking at the speedup figures, PRPQ can achieve 16% - 176% performance improvement over PRPD. The maximum achieved speedup of $2.67\times$ happens in the case of *Linear 0 evenly* dataset. This is due to the fact that the number of picked skew keys *hn* is always large and in sthis case, the key distribution at each node



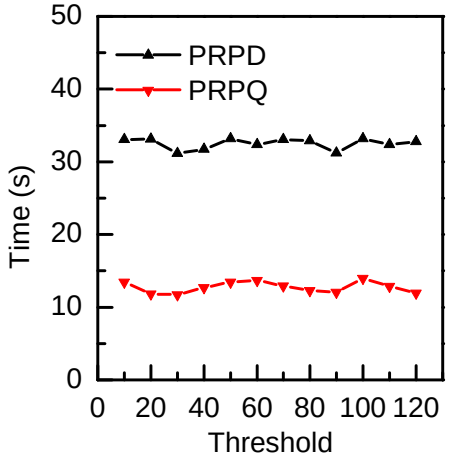
a) Zipf 1 evenly (128, 16, 1.16)



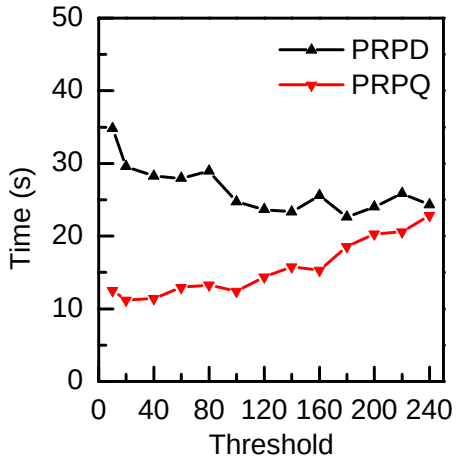
b) Zipf 1.4 evenly (4096, 32, 1.22)



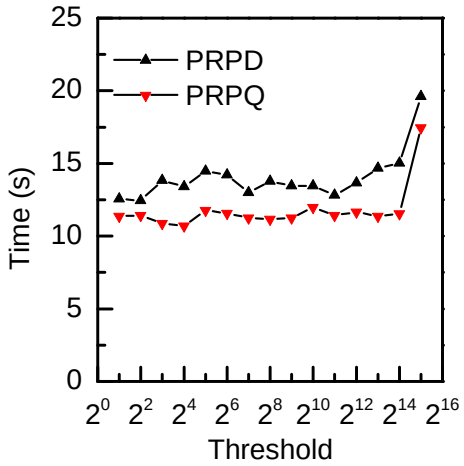
c) Zipf 1.4 range (1024, 512, 1.27)



d) Linear 0 evenly (90, 30, 2.67)



e) Linear 1 evenly (180, 40, 1.98)



f) Linear 1 range (4, 16, 1.16)

Fig. 7.4 Runtime of PRPD and PRPQ with increasing threshold t over different datasets ($64M \bowtie 1B$ with 192 cores).

follows $f(r) = 23170/192 = 121$, namely each key appears 121 times. Thus, when $t < 121$, all the $hn = 46341$ keys at each node will always be processed as skew keys, which makes the defined time difference ΔT_{24} large. This also appears in the cases Figure 7.4(a), (b) and (e): with a small t at the beginning, a large hn leads to a large ΔT_{24}. With increasing t, the difference decreases to 0 as hn becomes smaller and smaller. Finally, the variations of the results achieved for different t values are only minor for the PRPQ algorithm while those in PRPD change more sharply, demonstrating that our algorithm is less affected by the input parameters. Defining the t in a range that achieves better performance would require additional, more complex or costly operations, therefore, we can expect that our algorithm could profit more on performance than PRPD in real applications.

Cardinality Experiments

We also examine the speedups by varying the cardinalities of the two input relations. For the Zipf distribution, we create data sets in which both relations are half the default size (scale 0.5, namely $32M \bowtie 512M$) and double the size (scale 2, namely $128M \bowtie 2B$). We vary the threshold and record the best achieved runtime. Table 7.2 shows the results, which demonstrate that our algorithm can achieve higher performance irrespective of the input size.

Table 7.2 Speedup achieved by PRPQ over PRPD with varying the size of inputs (using 192 cores).

Skew	1			1.4		
Scale	0.5	1	2	0.5	1	2
Speedup	1.42	1.16	1.20	1.44	1.22	1.48

7.5.2 Network Communication

Communication costs are evaluated through measuring the number of received tuples at each place. The average number of received tuples is presented in Table 7.3. We can see that all four algorithms receive the same number of tuples when the dataset is uniform. This is reasonable, since there is no skew and all the tuples of PRPD and PRPQ are only processed by the *partial redistribution* while the number of query keys and returned values (both consider 1/2 tuple) is equal to the number of total tuples in *Query*. With increasing the skew, the number of the received tuples in *Hash* do not change, as all tuples still need to

be redistributed. In contrast, the other three methods show a significant decrease, as a large number of skewed tuples are not transferred in PRPD and PRPQ while *Query* groups the skewed tuples and only transfers the unique keys.

We also track the number of received tuples for different threshold t values and present the results in Figure 7.5. It can be seen that in PRPD the number first decreases and then increases, showing a trade-off between the number of duplicated and redistributed tuples. For PRPQ, the number of received tuples is always increasing, however, it is less than PRPD for each given t, a result that is consistent with the analysis in Section 7.3.3. Combining this with the value where best performance is achieved, t is set to 2^7 and 2^{14} for PRPD, values that are greater than the values of 2^4 and 2^5 for PRPQ respectively. This is the reason why PRPQ clearly transfers less data than PRPD in Table 7.3, notably $24\% - 54\%$ less under the skews.

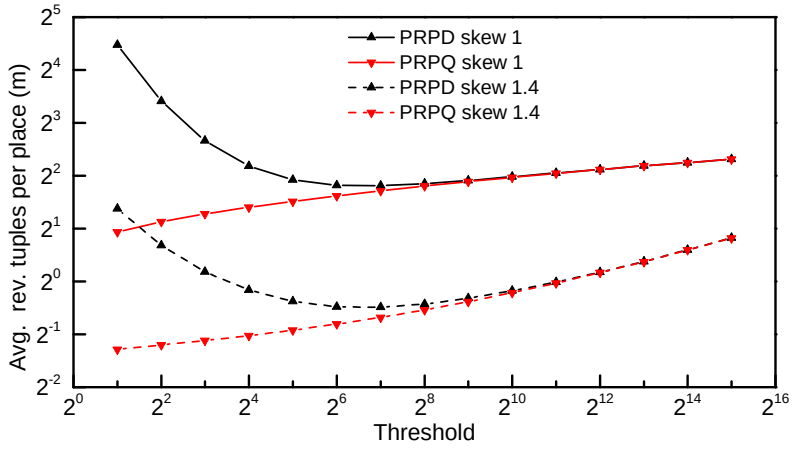


Fig. 7.5 Average number of received tuples at each place by varying the threshold ($64M \bowtie 1B$ with 192 cores).

7.5.3 Load Balancing

As previously discussed, the computation C_k has the same characterization as the communication N_k with regard to load balancing for each algorithm, therefore we evaluate this metric by the number of received tuples. The values for the maximum and average number of received tuples at each place are shown in Table 7.3 as well. We can see that all four algorithms achieve perfect load balancing when the data set is uniform. With increasing skew, the difference between the value of the maximum and the average for *Hash* increases, indicating poor load balancing in the presence of skew. In comparison, PRPD and PRPQ have more tolerance, showing their ability for handling the skew. The basic *Query* algorithm

is balanced, in line with our theoretical analysis in Section 7.3.3. We also note that the difference between the maximum and average values in PRPQ is smaller than that in PRPD under skews, manifesting slightly better load balancing. This is consistent with the analysis in Section 7.3.5: decreasing t can further improve the load balancing (as mentioned, t in PRPQ is smaller than PRPD).

Table 7.3 Detailed number of received tuples at each place (millions)

Skew/	**0**		**1**		**1.4**	
Algo.	**Max.**	**Avg.**	**Max.**	**Avg.**	**Max.**	**Avg.**
Hash	5.94	5.94	62.40	5.93	347.76	5.94
PRPD	5.94	5.94	3.53	3.51	1.16	1.13
PRPQ	5.94	5.94	2.65	2.64	0.53	0.52
Query	5.94	5.94	2.12	2.12	0.43	0.43

7.5.4 Scalability

We evaluate the scalability of our PRPQ implementation by varying the number of processing cores on the three default datasets, from 24 cores (2 nodes) up to 192. Results are presented in Figure 7.6, and each phase there is consistent with the implementation explained in Section 7.4.2.

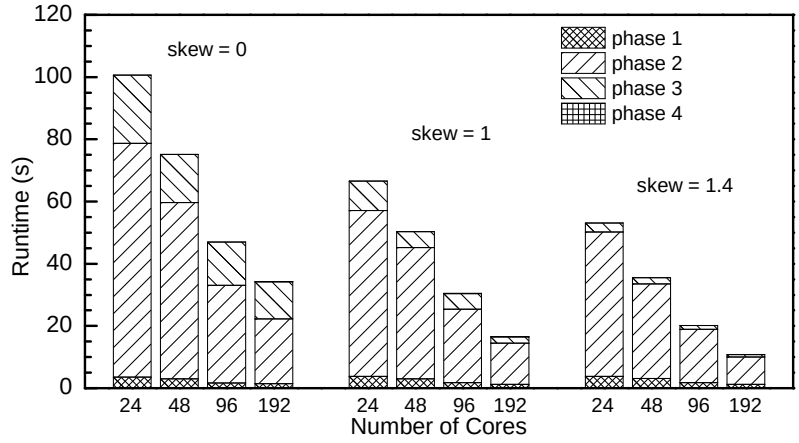


Fig. 7.6 The runtime breakdown of PRPQ under different skews by increasing the cores ($64M \bowtie 1B$).

It can be seen that PRPQ generally scales well under different skews. Notably, the relative speedup achieved between 48 and 96 cores is close to the ideal 2x, which is obviously

greater than that between other nodes. This could be attributed to the network overhead, that the network is extended at the beginning and the data set becomes comparably small for the underlying system when using 192 cores.

With details for each phase, under low skew, phase 2 and 3 scale well and are the dominant factor of the runtime achieved. In the case of high skew, the third phase becomes comparably much smaller and the second phase starts to dominate the performance, which decreases with increasing the number of cores. As the second phase mainly focuses on data transfer and the third, on join operations, the network load has a higher impact on the join performance than on the computation workload. For example, in the case of 192 nodes and high skew, the second phase takes 8.812 secs while the third takes only 0.739 secs (note that this also includes the time to push back the returned values). Finally, we note that cost of the fourth phase is extremely small and can therefore be ignored. The reason is that both the size of hash tables in T and the number of looked up elements (returned values) at each place relies only on the number of picked skew tuples, which is very small in our tests, resulting in a final lookup cost in the order of tens of milliseconds.

7.5.5 Comparison with Hash-based Joins

We conclude our analysis with a comparison with the commonly used *Hash* algorithm, by analyzing the performance improvement achieved for *joins* in each algorithm for different numbers of nodes.

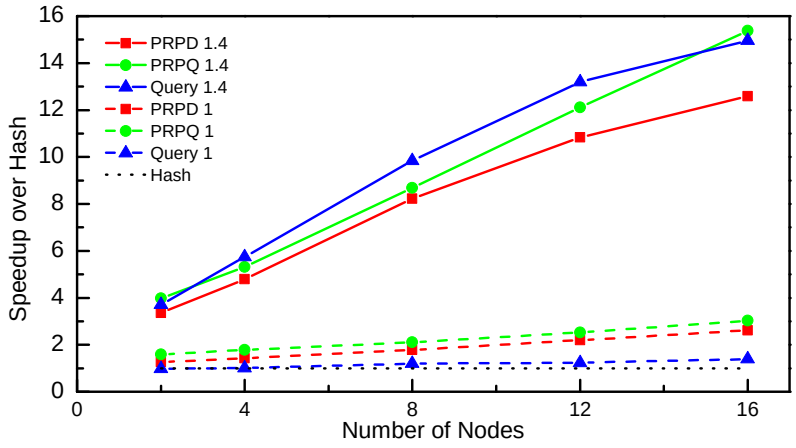


Fig. 7.7 Speedup ration over the hash algorithm under different skews by varying the nodes ($64M \bowtie 1B$).

Figure 7.7 presents the speedup ratio of PRPD, PRPQ and *Query* algorithm over the basic hash method with increasing number of nodes from 2 (24 cores) to 16 and for skew

values 1 and 1.4 respectively. All three algorithms consistently achieve speedups, demonstrating their ability to handle skew on distributed architectures. Furthermore, their speedups generally increase with increasing number of nodes as well as with increasing the degree of skew for a fixed number of nodes. Furthermore, for high skewed data, PRPQ achieves nearly linear speedup while PRPD and *Query* do not. This can be attributed to the following reasons: (1) For *Query*, the frequency of each element at each place decreases with increasing nodes, namely the ratio of low frequency elements increases. This in turn has a negative effect on speedup, as *Query* is not good at processing such low frequent data. (2) In comparison, via the variable t, PRPQ always knows *high-frequency elements by Query and low-frequency elements by Hash*. This presents an optimal way to process the data and achieves better speedups. (3) The broadcast cost increases with increasing the number of nodes, which results in scalability loss in PRPD.

7.6 Conclusions

In this chapter we have introduced a new approach for parallel joins which we have called PRPQ (*partial redistribution & partial query*), which targets high-performance, robust joins in distributed systems. We have conducted a theoretical analysis of our method and also presented a comparative quantitative evaluation.

The experimental results demonstrate that the proposed PRPQ algorithm is efficient and robust in the presence of different data skews. Moreover, the results associated with the theoretical analysis also highlight that our implementation always outperforms the state-of-art PRPD algorithm [127] under different join workloads.

We envisage that this new proposed join procedure can contribute to the performance of query executions in our framework. In the next chapter, we will detail the design of an efficient indexing structure, to meet the speed requirements on data loading and data querying of a large data analysis system.

Chapter 8

Fast Distributed Loading and Querying of Large RDF Data

8.1 Introduction

Fast loading speed and query interactivity is important for the exploration and analysis of RDF data in a web-based (large-scale) analytical environment. In such scenarios, large computational resources should be tapped in a short time, which requires very fast data loading of the target dataset(s). In turn, to shorten the data processing life-cycle for each query, exploration and analysis should also be done in an interactive manner.

In the previous chapters, we have proposed new approaches to improve the performance of RDF encoding and parallel joins. The former method has laid a solid foundation for data storage as the output encoded triples can be loaded as indexes directly. The latter technique can be used for SPARQL query executions with the results retrieved from the built indexes. It can be seen that the indexes play a pivotal role in system implementations, consequently, an efficient index targeting fast data loading and retrieving becomes critical. We next turn to the design of such an index for our system.

In Chapter 2, we divided the indexing approaches of existing distributed RDF systems into four types based on their data partitioning and placement patterns, namely similar-size partitioning, hash-based partitioning, sharded/partitioned indexes and graph-based partitioning. From the analysis of their advantages and disadvantages, we concluded that the techniques outlined operate on a trade-off between loading complexity and query efficiency, with the earlier ones in the list offering superior loading performance at the cost of more complex/slower querying and the latter ones requiring significant computational effort for loading and/or partitioning.

In this chapter, we propose efficient methods for processing RDF using dynamic data repartitioning to enable rapid analysis of large datasets. Our approach adopts a two-tier index architecture on each computation node: a lightweight primary index, to keep loading times low, and a series of dynamic, multi-level secondary indexes, calculated as a by-product of query execution, to decrease or remove inter-machine data movement for subsequent queries that contain the same graph patterns. We are applying a set of parallel techniques that *combine the loading speed of similar-size partitioning with the execution speed of graph-based partitioning* in our framework. The detailed elements of our approach are as follows:

1. We use fixed-length integer encoding for RDF terms and constant-time operations for indexing (i.e. indexes are based on hash-tables), so as to increase access speed.

2. During indexing, we do not use network communication, so as to increase loading speed.

3. We maintain a local lightweight primary index supporting very fast retrieval, to avoid costly scans.

4. We use secondary indexes supporting non-trivial access patterns that built dynamically, as a byproduct of query execution, to amortize costs for common access patterns.

5. We optionally reduce secondary indexes into filters, so as to to reclaim memory.

From that basis, we summarize the contribution of this chapter as follows: (1) We present a dynamic distributed RDF indexing that can both load data and compute queries fast on large RDF data, with a focus on analytical queries. (2) We implement our system with the X10 language and evaluate our system on a cluster using the LUBM benchmark [53]. (3) Experimental results show that our primary index results in very fast loading speeds: It takes only 7.4 minutes to load 1.1 billion triples on 16 nodes, for a throughput of 2.48 million triples per second, outperforming RDF-3X [89] by a factor of 53 and 4store [54] by a factor of 16. (4) Our secondary indexes significantly speed up computation, bringing the performance of our approach close to that of RDF-3X and 4store: It takes about 9, 4 and 0.4 seconds to execute the two most complex LUBM queries over our primary, 2nd level and 3rd level indexes respectively, outperforming RDF-3X in all cases, and 4store for the latter two cases. For the other queries and indexes, our approach still stays within an interactive response time. Additionally, our tests also show, by using filters, we can achieve 1.14 - 3.45x execution speedup, with minimal storage overhead (up to 1.2% of index size).

The rest of this chapter is organized as follows: In Section 8.2, we describe the detailed implementation of the data loading with the primary index building. We present the methods for secondary indexes building in Section 8.3 and approaches to generate the responsible distributed filters in Section 8.4. In Section 8.5, we evaluate a prototype of our implementation and compare to RDF-3X and 4store. In Section 8.6, we discuss both the techniques used in our design and the comparison with related work. Finally, in Section 8.7, we conclude the chapter.

8.2 Data Loading

The data loading process contains two parts, *triple encoding* and *primary index building*. Since the former implementation has been presented in Chapter 5, we just focus on the latter one. We refer to the primary index as (l_1), consequently the secondary indexes as 2nd-level (l_2), 3rd-level (l_3), etc. as in the following chapters.

After the parallel encoding implementation, we build the primary index l_1 for the encoded triples at each node. Similar to many triple stores, the index itself contains all the data. We use a modified *vertical partitioning* approach [3] to decompose the local data into multiple parts. Triples in [3] are placed into n two-column *vertical tables* (n is number of unique properties), which has been shown to be faster for querying than a single table. However, in [3], to efficiently locate data, all the *subjects* in each table are sorted, which is time cost ($Nlog(N)$) in terms of data loading, especially when the tables are huge.

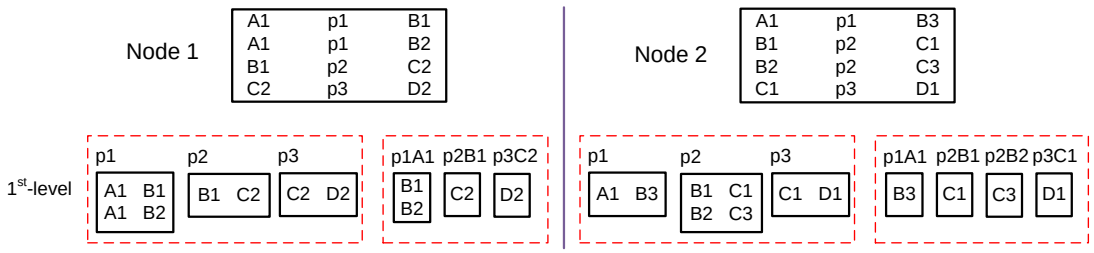
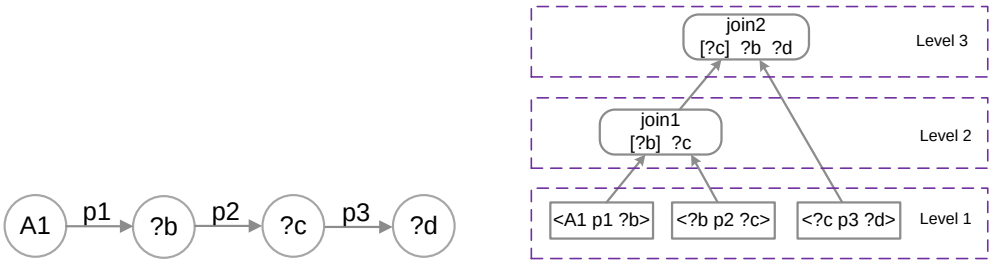


Fig. 8.1 The triples and the primary index for a simple two node system (vertical tables in the dashed square compose the $P \rightarrow SO$ and $PS \rightarrow O$ part of l_1).

In comparison to that, we only use linear-time operations for indexing, inserting each tuple in an unordered list in a corresponding *vertical table*. To support multiple access patterns, we build additional tables. By default, we build $P \rightarrow SO$, $PS \rightarrow O$ and $PO \rightarrow S$, corresponding to the most common access patterns. For example, Figure 8.1 shows the vertical tables of the primary index l_1, which is based on partitioning on the predicate and

the predicate-subject of each encoded triple at each node (note that the triples are in the form of integers in this step, we use the *string* format in our examples just for the sake of readability). As each node builds their tables independently, there is no communication over the network for this step. As we will show in the evaluation, local indexing is very fast, so we could support additional indexes, e.g. to support more efficient joins on the predicate position, with minimal impact on performance.

As in all RDF stores, there is an element of redundancy in terms of data replication. Our index consumes more space than the vertical partitioning approach in [3], or a compressed index approach such as the one found in [89]. Nevertheless, our focus is on speed and horizontal scalability, which increases total available memory. In addition, based on the fast encoding method described above, the build process of the primary index is very lightweight: (1) triples are encoded and indexed completely in-memory and all accesses are *memory-aligned*, reducing CPU cost; (2) there is no global index as we only build an index for local data on each computation node, *reducing the need for communication*; (3) we avoid sorting, or any non-constant time operation, meaning that the *complexity of our approach is* $O(N)$, where N is the number of local statements; and (4) the encoding algorithm achieves good load balancing, which translates to good load balancing for the (local) indexing. The above factors contribute to very fast indexing, as we will show in our evaluation.



(a) A SPARQL query. (b) An execution plan.

Fig. 8.2 An example of a simple SPARQL query graph and its query plan.

8.3 Data Querying

Once we have built the primary index, we can compute SPARQL queries through a sequence of lookups and joins. In this chapter, we only focus on queries consisting exclusively of basic graph patterns (BGPs), since filters, aggregations etc. have been studied in the literature in parallel databases for decades. Figure 8.2(a) shows a simple but common SPARQL query, which consists of three triple patterns that form a graph pattern. The evaluation for such a

query is to find all the solution mappings from variables in the three patterns to RDF terms of the underlying data. A reasonable execution plan for this query is shown in Figure 8.2(b). It has three lookup operators (one for each triple pattern) and two join operators, where the *bracket* indicates the join variable. In the following discussion, we assume that there is no join with predicate variables, since this is the most uncommon category of joins and our indexes could easily be adapted to efficiently support this (creating additional indexes would not significantly affect loading speed).

Parallel Hash Joins

In our system, with the primary index l_1, we can easily look up the results for a statement pattern at each node. For example, for the two-node system with the same triples as stated in Section 2.1.1, for the two triple patterns of *join1* in Figure 8.2(b), through looking up the vertical tables with properties $p1A1$ and $p2$, we can get the bindings for the variables $?b$ and ($?b$,$?c$) at each node:

	node 1	*node 2*
$?b$	$B1, B2$	$B3$
$(?b, ?c)$	$(B1C2)$	$(B1C1), (B2C3)$

This lookup process can be implemented in parallel and independently for each node. Nevertheless, a *join* between any two sub-queries cannot be executed independently at each node since we have no guarantee that join keys will be located on the same node. We adopt the parallel hash-join implementation in our system. Namely, results of each subquery are redistributed among computation nodes by hashing the values of their join keys, so as to ensure that the appropriate results for the join are co-located [120]. For *join1*, we redistribute all results of the first two triple patterns by hashing bindings for the variable $?b$, and then implement the local joins for the received terms at each node. The detailed flow of this process is shown in the first two segments of Figure 8.3. Similarly, *join2* is based on hashing the bindings of $?c$, for both the intermediate results from *join1* and the results of the third triple pattern.

Secondary Indexes

The local lookup for each triple pattern at each node is very fast, since we only need to locate the corresponding index table in l_1, and then retrieve all the elements. E.g. for the pattern $<?b\ p2\ ?c>$, we can find the vertical table p_2 and return its results in constant time (since we are using hashtables to index terms in the partitioned tables).

For join operations, as we have to redistribute all results for each triple pattern as well as the intermediate results, data transfers across nodes become costly, in terms of bandwidth and coordination overhead. Although many SPARQL queries with deep join trees generate large volumes of intermediate results anyway, compared to the hash- or graph-based partitioning approaches, for the lower levels of the join tree, our approach would transfer much more data. Therefore, efficient strategies are required to minimize data movement and improve query performance.

Algorithm 18 Query Execution and Secondary Index Building

The primary index l_1 has been built, let $\mathbb{Q}$ be a query queue to be processed, l the secondary indexes initialized as $\emptyset$ at each node, r the intermediate results to be joined initialized as $\emptyset$.

```
Main procedure:
1: for each Q ∈ ℚ do
2:     r=plan(Q)    //Plan query with root r
3:     compute(r)
4: end for

Procedure compute(n):
5: r_i = l.lookup(n)
6: if r_i ≠ null then
7:     return r_i    // If an index already has the result
8: else
9:     for each child c in n parallel do
10:        if c is a triple pattern then
11:            lr_i=l_1.lookup(n)
12:            r_c=redistribute(lr_i)
13:        else
14:            r_c=compute(c)
15:        end if
16:        r.add(r_c)
17:        if isIndexable(r_c) then
18:            l.index(c,r_c)
19:        end if
20:    end for
21:    return join(r)
22: end if
```

We build secondary indexes (l_2 ... l_n), based on the redistribution of data during query execution. The build process of such indexes is presented in Algorithm 18. We have a queue of queries $\mathbb{Q}$. For each query Q, we assume a planning method (which is beyond the scope

of our work) that results in an execution plan represented as a tree with root r. We assume that queries in the queue are processed sequentially and each node keeps a set of indexes of various levels $l_{1..n}$. All nodes start with index l_1 built and all other indexes are empty.

We evaluate the expressions in the tree bottom-up, in parallel (lines 9 and 14), redistributing results as required (line 12). The function `isIndexable()` determines whether nodes should retain the (indexed) data from remote nodes. An example criterion for indexing is the depth of the subtree (corresponding to the indexing level described in the following), which we also use in our implementation. The construct `parallel do` implies synchronization at `end for`. Results from existing indexes are re-used when possible (lines 6 and 7). Once the results of all children of a node become available, a join is executed. Note that this process implies a high degree of parallelism since individual joins are executed in parallel and multiple join expressions are calculated in parallel, when possible.

An example for the query in Figure 8.2 over the data in Figure 8.1 is shown in Figure 8.3. In this figure, for the pattern $<?b\ p2\ ?c>$ in *join1*, we get its results from the table $p2$ of l_1 at each node, then redistribute all of them on the value of ?b. In this way, the tuple $<B1\ C1>$ at node 2 is sent to node 1 (based on hashing $B1$) and the tuple $<B2>$ is sent to node 2. Joining can now be done locally.

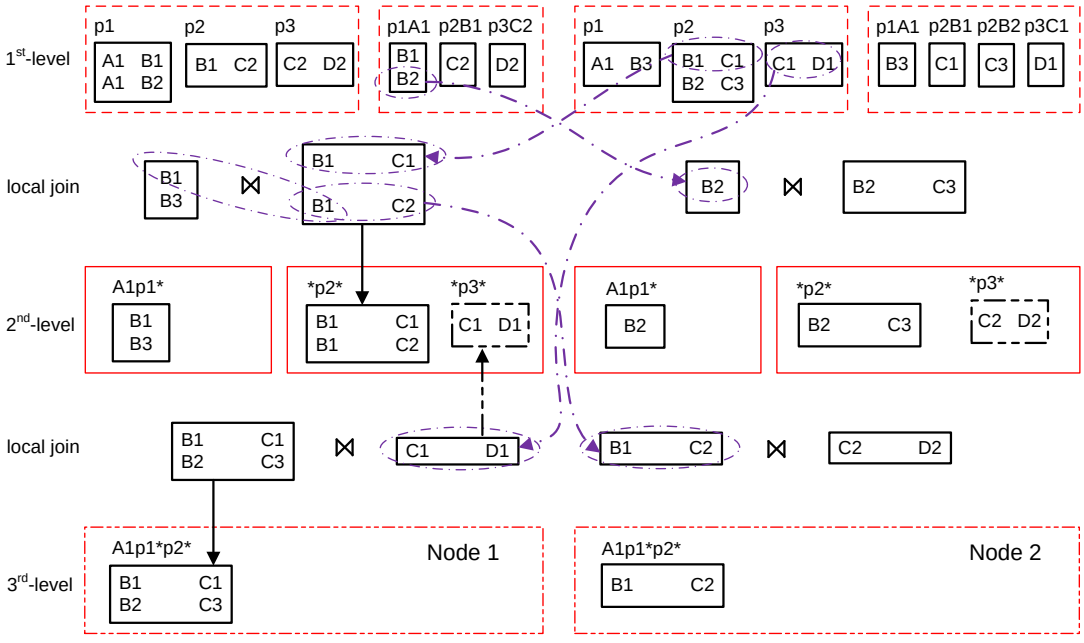


Fig. 8.3 Example of query execution and secondary index building.

After redistributing results, a set of new tables is built on l_2: for $*p2*$ and $A1p1*$. The *join2* is executed in a similar manner: since there is no table for $*p3*$ in l_2, the data from

l_1 is redistributed. The results for *join1* are already available, but need to be redistributed, since we are now joining on a different variable $?c$.

After this redistribution, the result can again be performed locally. Two new tables are generated: $*p3*$ on l_2 and $A1p1*p2*$ on l_3.

As shown above, the index is constructed by a simple *copy* of the redistributed data, which is introduced by a *join* of a query. Namely, it is a byproduct of query execution. The secondary indexes are re-used by other queries that contain patterns in common. Note that we are using the term indexing instead of caching, because the data is re-partitioned on demand and is fully indexed in a sharded manner, as opposed to storing intermediate results and re-using them, such as the *cache* used in centralised RDF stores [114, 115]. In practice, this means that, indexes can be re-used for any query containing them. The consequent cost is that we need to re-compute the joins locally for every query. In principle, we could combine this approach with caching.

Index Levels

Based on the example above, we define the *stratified semantics* of the query execution plans in our system.

Definition 1. Given an execution plan containing a finite set of general operations, which may have both *lookups* and *joins*. The *level L* of an operation is assigned to L_k ($k \geq 1$) in the plan as follows:

- all triple pattern lookups are L_1 operations and
- an operation with L_{k+1} consists of a *join* between a set of L_i ($i \leq k$) operations, where there exists at least one $i = k$ and the join variable in L_{k+1} and L_k is different.

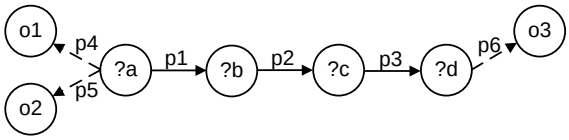
For example, in the execution plan in Figure 8.2, three lookup operations are located in L_1, *join1* is assigned to *Level 2* because both its inputs are L_1 operations, and consequently *join2* is an L_3 operation.

According to Algorithm 18, with a built second-level index l_2, the execution of a join located in L_2 will be cost-free in terms of network communication. This means that, there are only local joins for the queries that contain only L_1 and L_2 operations, for some given query patterns. However, for a query containing higher level operations, we still need to transfer the intermediate results to remote nodes. An l_3 index allows executing L_3 operations with no network traffic, and so on.

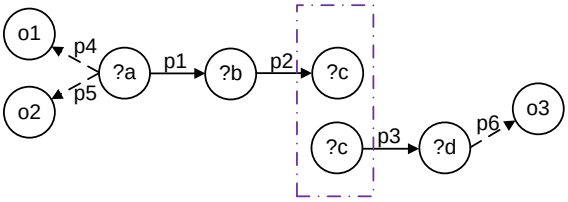
In the process of building the k-th level index l_k, *if we run all possible queries, what will the data on each node look like*? In fact, according to the terminology regarding *graph partitioning* used in [61], the 2nd-level index in our method on each node will construct a 2-hop subgraph, the 3rd-level one will be a 3-hop subgraph, and l_k will be a k-hop subgraph. This means that our method essentially does dynamic graph-based partitioning starting from an initial equal-size partitioning, based on the query load. Therefore, our system can combine the advantages of fast data loading and efficient querying. We will show that this design is indeed efficient in our evaluation presented in Section 8.5. In addition, the theoretical results from [61] can be applied for our approach as well.

8.4 Distributed Filters

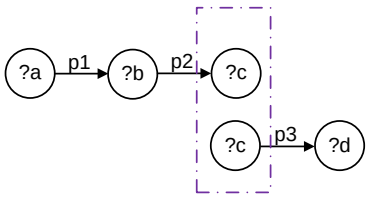
Secondary indexes l_k reduce the network communication for a query. As k increases, the transferred data between nodes decreases, resulting in improved performance. However, the space for the entire index l also increases, constituting a trade-off between space and performance. In the meantime, the higher level an index is, the larger its size could be. There are two main reasons for this: (1) l_{k+1} has more tables than l_k, as its properties are constructed by combining the properties of l_k with other indexes'; and (2) the size of a table in l_{k+1} could be larger than that in l_k, for example l_{k+1} could include the *cartesian product* of elements in l_k.



(a) A query graph.



(b) *join* in its *full* path.



(c) *join* in its *main* path.

Fig. 8.4 A complex SPARQL query graph and the *join* in its graph path.

Various strategies can be applied to reduce the size of l, such as reuse of repeated parts between each index, building indexes for frequent graph patterns etc. [67]. Orthogonally

to these approaches, we introduce *distributed filters* as a compact alternative to secondary indexes.

Because some elements taking part in a join could not possibly have a contribution to the output, if we know their join results (for example as intermediate results from a previous query), we can remove such elements before performing the join. Filters operate by filtering out intermediate results at the source, based on aggregate results stored during previous query executions and are focused on a join point (e.g. the join on ?c in Figure 8.4(a)), or, otherwise expressed, an operator on the query plan.

For a join on v between two graph patterns of a query s_i and s_j, we propose one of the following: (1) *full path filters*, which contain all values of v after the join of s_i and s_j, and (2) *main path filters*, containing the results of v over the join between the main path graph of s_i and s_j. The main path graph is defined as the graph consisting only of the triples with a single constant (most commonly the predicate).

For example, the query graph shown in Figure 8.4(a), contains a join between a L_3 and L_2 operation on ?c. The join results of ?c in Figure 8.4(b) can be used instead of an l_4 table, constituting a F_4 filter. Bindings for c are filtered at source according to this filter, *for this particular query or any query that subsumes this query*. Figure 8.4(c) shows the structure of a main path filter for the same query.

These two filters have their own advantages and shortcomings: (1) the full path filter can remove more redundant elements. It is also a byproduct of query implementation and can be applied to our system by projecting out the desired variables and eliminating duplicates. However, the usage of such a filter is limited to queries that subsume the entire pattern; (2) the main path can be used in more queries, especially since it corresponds to the less selective part of the query, essentially capturing the structure of the graph. Regardless, like many other path filters [76], it can only remove part of the redundant elements, since it is less selective than the query. In addition, it needs pre-execution (or a less efficient query plan starting from the non-selective part).

For a specified query, obviously, the size of a filter will be much smaller than the corresponding indexes, because we only store the (discrete) results for the join variable. In the meantime, we should also notice that a filter is less efficient than a index, as it only *reduces* the network communication, while the index can *remove* such communication. In addition, the filtering operation bears some, relatively small, computational cost. We will compare the performance of both techniques in our evaluation in Section 8.5. In fact, we can adopt a hybrid construction for the high-level index, e.g. using index architectures for some join patterns while applying filter architectures for others, so as to achieve the best balancing be-

tween performance and space. In addition, for full path filters, it is easy and computationally inexpensive to *reduce* an index to a filter. Finally, we should note that filters are partitioned in the same way as secondary indexes across the network, placing an equal space burden on all nodes.

8.5 Evaluation

We present an experimental evaluation of our approach to determine the performance of our lightweight indexing, the secondary indexes and the distributed filters. We run the LUBM [53] benchmark over a commodity cluster and compare loading speed and performance of query execution with a top-performing RDF store running on a single node as well as a cluster RDF store.

The evaluation platform is the same as the described previously. We use 16 IBM iDataPlex® nodes with two 6-core Intel Xeon® X5679 processors clocked at 2.93 GHz, 128GB of RAM and a single 1TB SATA hard-drive, connected using Gigabit Ethernet. We use Linux kernel version 2.6.32-220, X10 version 2.3 compiling to C++ and gcc version 4.4.6.

8.5.1 Setup

Although in this chapter we are focusing on an indexing method, as opposed to a full clustered RDF store, we have performed comparisons on query execution with RDF-3X [89] and 4store [54]. The former represents the state-of-the-art in terms of a single machine stores while the latter is a clustered RDF store designed to operate main in-memory[1]. We have modified the setup so as to isolate the BGP processing costs and nullify, to the extent possible, the advantage of our approach and 4store regarding I/O performance. Specifically, we do not count the time spent on *query parsing*, *plan generation*, *dictionary lookup* or *result output*, so as to focus on analyzing the core performance of query execution. More exactly, we only report times for the operations of *index location*, *index scanning* and the relative *joins* in the execution phase. To achieve this:

- For RDF-3X, we deployed version 0.3.7 on a single node[2]. We add responsible *profiling counters* to the source code as the same as the ones in Chapter 3.

[1] Refer to http://4store.org/trac/wiki/Tuning

[2] Note that we use single-node RDF-3X as this open-sourced triple store is popular used as a performance reference for RDF stores, and also for cluster solutions such as [61].

- For 4store, we installed the latest version 1.1.5 on 16 nodes. We use the default indexes and set the value of the system parameter *segment* to 256 as a recommended value. In the meantime, we also set *soft-limit* to -1, so that we can retrieve all the results. As 4store has provided the desired profiling tools, we can directly get the time taken to locate the results and perform the joins on the backend systems (storage layer), through examining the *bind time*. We have confirmed this with the 4store community.

We do not compare MapReduce-based approaches since, due to platform overhead, they do not execute interactive queries in reasonable time. For example, SHARD [97], has run-times for LUBM in the hundreds of seconds.

8.5.2 Benchmark

We load LUBM(8000), containing about 1.1 billion triples (about 190GB) and run all 14 queries on this data. As our system does not support RDF inference, we use a modified query set to get results for most queries, which is given in Appendix B. For example, since the basic graph pattern <?x type Student> returns no results in Query 10, we use <?x type GraduateStudent> instead.

We have chosen LUBM because our system currently only supports BGPs and because LUBM includes BGPs with varying selectivity and complexity. Although there exist well-known weaknesses in this benchmark, such weaknesses mainly affect query planning and data distribution, which is not the focus of our system. As future work, we are planning to experiment with more complex benchmarks, in terms of structure and features (such as aggregation).

To conduct a precise performance comparison, we load and query data on memory, so as to reduce the effect of I/O. Therefore, we set the index locations of RDF-3X and 4store to a `tmpfs` file system resident in memory at each node, so that queries can be fully implemented over distributed memory. For data loading, because our `tmpfs` file system at each node cannot hold all 1.1 billion triples, we load data from hard-disk to memory for the two stores. Although our system can operate completely in distributed memory, in the interests of a fair comparison, we read data from disks as well during the data loading process.

8.5.3 Data Loading Time

We load 1.1 billion triples and build three primary indexes (on P, PO and PS). For RDF-3X and 4store, we report the time to load data from disk into the memory partition(s). For both

systems, we are using the default indexes.

Table 8.1 Time to load 1.1 billion triples

RDF-3X	23296 seconds
4store	7078 seconds
Our system	Read from disk: 103 seconds Triple encoding: 254 seconds Building l_1: (P, PO, PS) 86 seconds Total: 443 seconds

As shown in Table 8.1, we take 103 seconds to read the data into memory, 254 seconds to encode triples and 86 seconds to build the primary index l_1, for an average throughput of 429MB or 2.48M triples per second. In comparison, 4store takes 7078 seconds, for an average throughput of 155K triples per second. The reason is that our loading process is totally in parallel at each core and our indexes are very lightweight, while 4store needs to do global sorts and uses a master node for coordination.

We also see that RDF-3X takes about 6.5 hours, for an average throughput of 47K triples per second, performing much worse than the other two systems (presumably because we are running on one node and because of the heavier indexing scheme of RDF-3X). From the results reported in [61], the graph-based partitioning method is even slower than RDF-3X, highlighting the advantage of our approach, in terms of loading speed.

8.5.4 General Query Performance

We execute all LUBM queries using l_1 and l_2, since the number of joins in most queries is small. Although our system does not use a cache as such, one could consider executions with secondary indexes as warm runs and l_1 as a cold run (we explain further the costs and benefits of additional index levels later in this section).

Table 8.2 shows the execution time for each query. Both RDF-3X and 4store are very fast for most queries, staying under 1ms, since many queries in LUBM are very simple. There is only a marginal difference between cold and warm runs, since we are operating in-memory. In our system, the execution over l_2 is generally much faster than over l_1, which shows that query performance can be highly improved by building a secondary index. The lowest speedup is achieved on Q2, Q9, Q6 and Q14, the reasons being that (1) Q2 and Q9 contain the L_3 operations (as defined previously), hence intermediate results still need

Table 8.2 Execution times for the LUBM queries over RDF-3X and 4store with cold and warm runs, as well as our system with the primary index l_1 and second-level index l_2 (ms)

Q.	RDF-3X		4store		Our system		#
	cold	warm	cold	warm	l_1	l_2	Results
1	0.19	0.17	9	8	500	14	4
2	11303	11217	4635	4510	8244	3917	2528
3	0.26	0.25	24	22	1635	20	6
4	0.34	0.28	0.45	0.32	10597	445	10
5	0.22	0.18	4.08	3.57	1012	13	146
6	409	382	6.49	5.71	12	12	20 mil.
7	0.64	0.54	0.19	0.15	8129	731	0
8	1.73	1.55	0.69	0.64	5145	564	1874
9	10253	9803	18148	17972	9533	4173	0
10	0.21	0.17	5.76	4.79	986	15	4
11	0.21	0.17	1.24	1.20	505	13	0
12	125	124	0.24	0.20	1285	384	125
13	202	199	18.49	16.01	1141	18	19905
14	1147	1055	21.19	20.45	16	16	63 mil.

redistribution over a l_2 index; and (2) Q6 and Q14 contain only a single triple pattern, thus l_2 is not built.

Comparing the warm run of RDF-3X and our implementation with the 2nd-level index: (1) our approach is slower than RDF-3X for simple and selective queries such as Q1 and Q3. RDF-3X uses some hundreds of μs to finish the operations of lookup and joins for candidate results while our system (also 4store) has to do synchronization over a distributed architecture, which has an overhead of about 10 *ms*, which is still acceptable; (2) our system is much faster at *complex queries*, for example Q2 and Q9, as we can implement joins in parallel; and queries having *low selectivity*, for example Q6 and Q14, since it has higher aggregate I/O; or possible both reasons, such as Q13.

Meanwhile, compared to 4store, we are generally slower, such as for the Q1, Q5, Q6, Q10, Q11 and Q13. Regardless, the difference in the time cost is very small, only of the order of *ms*. The possible reason could be the overhead of our implementation, because we only adopt *hash join* as local joins in our method and we have to build the hash table first and then implement probings. We are also slower on Q4, Q7, Q8 and Q12, in the order of *100 ms*, it could be that 4store optimizes the coordination between each node. However, this difference is still acceptable as our approach is still suitable for interactive applications (in *ms*).

Considering the most complex queries Q2 and Q9, we are obviously much faster[3], of the order of *sec*. Moreover, we can further improve our performance by employing higher level indexes. On the other hand, our method is also faster on the simple queries Q3 and Q14. The reason for this could be that: (1) for Q3, the used *hash join* performs very fast in the presence of small-large table joins, and (2) for Q14, we can quickly locate the required indexes and then organize scans for a large number of tuples.

Most importantly, it should be highlighted that we have parallelised all the operations and distributed everything, in terms of data loading (including triple encoding and primary indexing building) and data querying (including joins, secondary indexing building and filtering). For LUBM, our approach is at least an order of magnitude faster at loading data while still keeping query response time within an interactive range.

8.5.5 Indexes and Filters

We examine the time cost to build indexes and filters, and examine query performance on executing Q2 and Q9, which are the most complex queries. We first build the second-level and third-level index for these two queries and then replace the third-level index by either the *main path* or the *full path* filter.

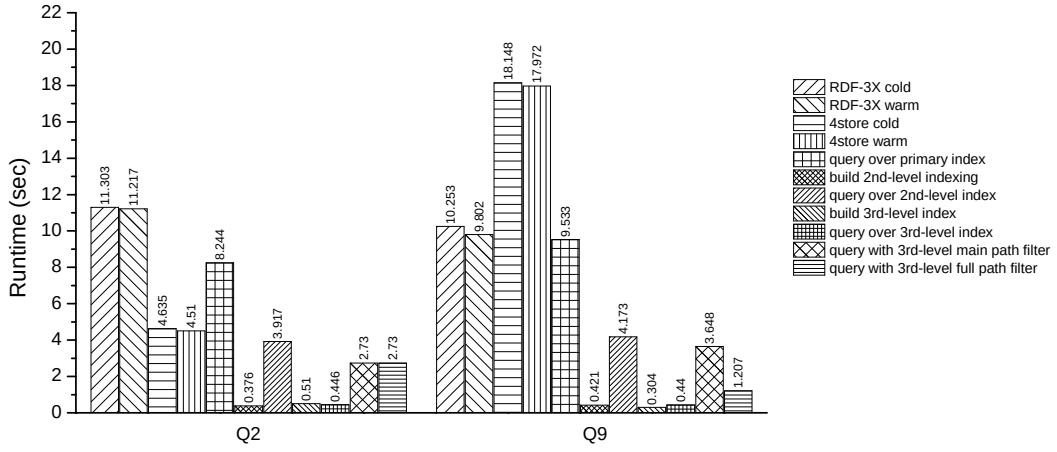


Fig. 8.5 Runtime for RDF-3X and 4store, and detailed runtime of each implementation for our system (over Q2 and Q9 using 192 cores).

[3]For 4store: (i) for Q2, the whole query time is 335 seconds; and (ii) for Q9, here we only provide the results when running without system parameter *soft-limit*. If we set this parameter to -1, the execution time is more than 7 hours.

Figure 8.5 shows that building a high-level index takes only hundreds of *ms*, which is extremely small compared to the query execution time. This operation is very fast, since it only involves indexing using in-memory hash tables. We can also see that, the higher the level of index is, the lower the execution time. For example, with l_3, Q2 and Q9 can be executed in 0.45 seconds, which is much faster than l_2 and RDF-3X. The reason is that, for l_3, there is no data movement between nodes for joins. Figure 8.5 also demonstrates that, with a filter, query execution time is higher than with l_3 (because we still have to redistribute elements over the network), but faster than with l_2 (speedups from 1.14 to 3.45), showing a trade-off between space and performance.

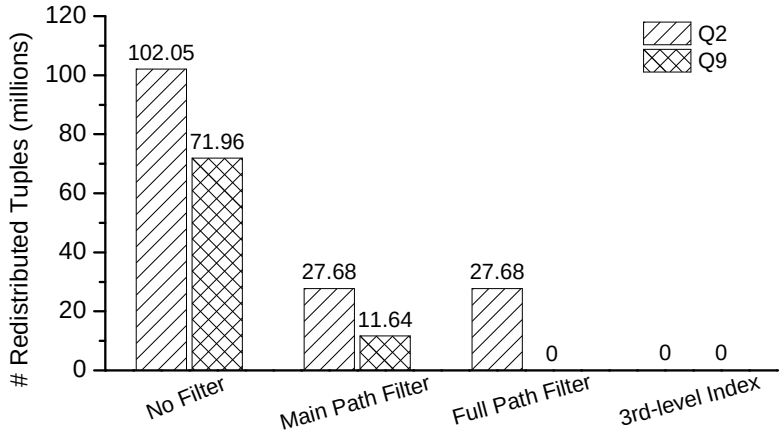


Fig. 8.6 Number of redistributed tuples.

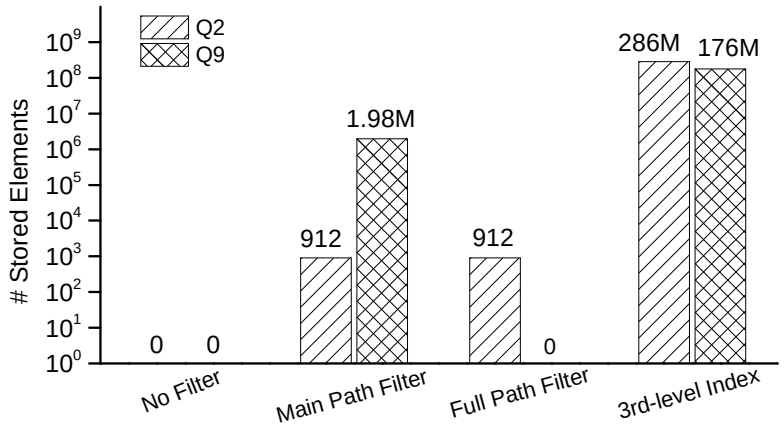


Fig. 8.7 Number of elements in a filter (index).

To further investigate the effect of filters, we record the total number of received elements and compare to l_2 without a filter. The results are presented in Figure 8.6, showing

that network communication can be greatly reduced through filters - a finding further supported by the performance improvement evident in Figure 8.5. With a *main path filter*, about 70% of data movement associated with Q2 is eliminated. However, we observe that the transfer time is reduced by only 37%, the reason being the communication overhead and the computational cost of filtering. Figure 8.6 shows that a *full path filter* can sometimes be better than a *main path filter*. For example, network communication is 0 when Q9 is using a *full path filter*, much less than that of the *main path filter*. In contrast, both filters perform the same for Q2. For a full l_3 index, network communication is zero.

In Figure 8.7, we show the space overhead for each option (expressed as elements, represented by long integers). We see that, for either method, the cost is much smaller than the number of tuples transferred without a filter (a maximum of 2 million elements stored compared to a minimum of 72 million triples transferred). For comparison, we also include the space cost for l_3 (it can be also seen that the size of a filter is up to 1.2% of the index size).

8.5.6 Load Balancing and Scalability

Because data skew is very common in RDF data [78], we measure load distribution across nodes on Q2 and Q9. We execute both queries over the primary index using 192 cores by recording the number of received elements on each core. As shown in Table 8.3, for the two redistributed operations in each query, there is nearly no skew in Q9. In contrast, there exists obvious skew in Q2, which indicates that skew-handling techniques such as the ones presented in Chapter 6 and Chapter 7 can be applied in our system to further improve the performance for such queries.

Table 8.3 Number of received tuples at each core (millions) for 192 cores

# received elements	**L1**		**L2**	
	Max.	**Avg.**	**Max.**	**Avg.**
Q2	0.987	0.871	0.801	0.532
Q9	1.595	1.593	0.377	0.375

We also test the scalability of our implementation by varying the number of processing cores. We run Q2 and Q9 over the second-level index and double the number of cores from 12 (a single node) all the way to 192. The results are presented in Table 8.4. It can be seen that the execution time of both queries decreases with increasing the number of cores. Nevertheless, both queries reach a plateau at around 4 seconds. The reason for this

Table 8.4 Runtime by varying the number of cores over 2nd-level index

# nodes	12	24	48	96	192
Q2	20.804	15.613	13.027	6.827	3.917
Q9	11.453	9.516	7.908	5.272	4.173

is that overhead starts dominating the runtime. With 192 cores, for each core, there will be approximately 191 (one from each other node) messages, with the associated coordination overhead, for a total of 532K and 375K tuples transferred for Q2 and Q9 respectively. As future work, we will work on methods to reduce the distribution for small indexes, so as to avoid this messaging and coordination overhead.

8.6 Discussion

We position our work against related work in *batch processing oriented* RDF systems, (clustered) triple stores, *graph partitioning based* systems and other literature from the database community.

RDF processing systems geared towards batch processing [79, 120] are based on architectures developed for a similar-size data partitioning model. In this respect, these systems are similar to the one proposed here in terms of fast data loading and minimal or no preprocessing. However, they execute queries directly over the *raw* data without any encoding process or additional index, resulting in a heavy network communication costs for complex queries and significant startup overhead. For example, while [79] can process massive datasets with zero loading time, its minimum runtime is in minutes, not seconds.

Systems such as SHARD [97] and the one in [62] generally adopt hash-based partitioning techniques. This leads to slower loading of RDF data, e.g. 0.5 hour to load 270 million triples is reported in [61]. These systems are similar to our system using the 2nd-level index. Therefore, they can avoid communication for simple queries containing only L_1 operations. For complex queries with higher-level operations, our system is much faster, because large amounts of data in these systems still needs to be redistributed across the network to perform joins.

Clustered RDF stores such as Virtuoso Cluster [44], BigData [112], YARS2 [56] and 4store [54] distribute indexes (typically SPO, POS etc.) over nodes in a cluster to improve I/O and join throughput. They are more similar in operation to single-node RDF stores than to our approach, offering lower loading speeds but also persistence and more space-efficient

indexing. As shown in our tests, we are much faster than 4store in data loading and also outperform it for complex queries.

Systems using graph-based partitioning such as the ones in [61, 128, 129], are similar to the ones using high-level indexes proposed here, which impacts positively on query performance. However, graph partitioning and triple placement in these systems happens at indexing time, hampering loading throughput. For example, the system described in [61] takes 4 hours to assign 270 million triples according to a 2-hop construction. Although [129] stores data as a graph, time spent on graph partitioning will still increase exponentially with increasing either the size of a graph or the parameter *hop*, because the connections between vertexes becomes more complex. In contrast, our system has no such costly operations, but organizes the sub-graph dynamically. Moreover, our incremental indexing process has proven to be very lightweight, requiring only hundreds of *ms*, in addition to query execution time.

Database cracking [64, 65] is an adaptive indexing technique that incorporates continuous self-organization of data storage based on selections in incoming queries. This idea has influenced the design of the secondary index used in the system described here. However, research on cracking is concentrated on incrementally sorting the raw data on a single machine, so as to reduce data lookup time. In comparison, we focus on reducing network communication and apply some concepts behind cracking to distributed systems and RDF data. Additionally, as data in our indexes is unordered, we can also apply the existing cracking methods to our local index, so as to further improve the final query performance of our system.

Result recycling refers to re-using intermediate results from past query executions. We apply a similar approach to [67], examining caching in a column store using an operator-at-a-time architecture. The main differences with our approach is that (1) we apply this on a distributed setting, (2) we store the remote data rather than the materialization of the intermediate results (i.e. we re-execute the joins locally) and (3) we apply this on RDF data, using indexing structures with different characteristics. In fact, the eviction and retention strategies in [67] could be adapted to our system.

Path-based filters proposed for semi-structured data [124] and RDF data [76] can efficiently identify and then reduce elements participating in joins by pre-joining sub-graphs. The two filters proposed here are similar to those in [76], which filter only for the unique items of a *join*. However, there exist four main differences: (1) the height of filters in [76] is limited, as they pre-join all the possible sub-graphs, which is extremely costly both in terms of time and space. In comparison, our *full path filter* is a byproduct of query processing and

filters are only built for the specified *join point*; (2) [76] applies filters on a single machine and focuses on techniques to reduce the size of filters while we use partitioned filters in a distributed system and have much less pressure in terms of space. We can incorporate the techniques in [76] to reduce the size of local filters; (3) filters in [76] are mainly used to reduce the time to lookup underlying sorted data, while we focus on reducing inter-machine communication; and (4) the structure of filters in [76] is similar to our *main path filter*, which can be less efficient than our *full path filter* for some queries, as shown in our experiments.

8.7 Conclusion

In this chapter, we present a distributed RDF data processing method designed for fast loading and querying over large-scale data. Based on a simple similar-size data partitioning infrastructure, we propose a dynamic two-tier index architecture and introduce the design of a pair of performance-enhancing distributed filters. Our implementation is tested with the LUBM benchmark [53] and the experimental results demonstrate that our approach can load data at least an order of magnitude faster than a clustered store operating in RAM while remaining within an interactive range for query processing and even outperforming state-of-the-art systems for expensive queries.

Chapter 9

Conclusions and Future Work

With the continuously increasing spread of the semantic web, an efficient RDF data management systems is becoming critical for the development of application which seek to take advantage of big RDF data. In this thesis, we have introduced a scalable analysis framework with several novel techniques for efficiently processing large scale RDF data. Here, we summarize the contributions of this thesis and suggests directions for future work arising from the developments undertaken.

9.1 Summary of Conclusions

Systemic Evaluation of RDF Stores

As RDF stores have developed, there have been various benchmarks and experiments that have attempted to evaluate the response time and query throughput of individual stores to show the weaknesses and strengths of triple store implementations. However, these evaluations have primarily focused on the application level and have not sufficiently investigated system-level aspects to discover performance inhibitors and bottlenecks. In Chapter 3, we proposed metrics based on a systematic study of the impact of triple store implementation on the underlying platform. We chose some popular triple stores as use cases, and performed our experiments on both a standard and an enterprise platform. Through detailed time cost and system consumption measures of queries derived from the Berlin SPARQL Benchmark (BSBM), we described the dynamics and behaviors of query execution across these systems. The collected data provides insight into different triple store implementation as well as an understanding of performance differences between the two platforms. The results allow us to identify performance bottlenecks in existing triple stores implementations which is useful

in our design efforts for large RDF data processing.

Efficient Parallel Hashing for Massive Data

Since hash tables are commonly used in high performance data analysis systems, we focused on investigating efficient parallel hash algorithms for processing large-scale data in Chapter 4. Currently, hash tables on distributed architectures are accessed one key at a time by local or remote threads while shared-memory approaches focus on accessing a single table with multiple threads. In comparison, a relatively straightforward "bulk-operation" approach seems to have been neglected by researchers. Based on such a method, we have introduced a high-level parallel hashing framework, *structured parallel hashing*, targeting efficiently processing massive data on distributed memory. We presented a theoretical analysis of the proposed method and described the designs of our hashing implementations. The analysis with evaluations have shown that the proposed method can vastly outperform distributed hashing methods and can even offer performance comparable with approaches based on shared memory supercomputers which use specialized hardware predicates. Moreover, we also characterize the performance properties of our hash implementations through extensive experiments, which allows us to make a more informed choice for our high-performance implementations.

Scalable RDF Compression

For interoperability, the semi-structured data elements in the semantic web are represented by long strings. This representation is not efficient for the purposes of semantic web applications that perform computations over large volumes of information, and the use of dictionary encoding for this purpose is particularly prevalent in RDF database systems. In contrast to most of the centralized implementations, in Chapter 5, we have proposed a scalable and efficient dictionary encoding scheme for encoding large RDF data over distributed architectures. We have described the detailed implementation and optimization of our algorithm using the X10 language and evaluated its performance on a cluster of up to 384 cores with datasets of up to 11 billion triples. The experimental results have demonstrated that our method is computationally fast and has achieved very high throughput in terms of RDF encoding. Further, when compared to the state-of-art MapReduce algorithm, we demonstrated a speedup of $2.6 - 7.4\times$ and excellent scalability.

Efficient and Robust Parallel Joins

The performance of joins in parallel database management systems is critical for query implementations. As data skew naturally exists in many applications, poorly engineered join operations always result in load imbalance and performance bottlenecks. State-of-the-art methods designed to handle such a problem offer significant improvements over naive implementations, however, performance could be improved further through removal of the dependency on global skew knowledge and the redundancy on data movements.

In Chapter 6, we introduced *query-based joins*, a novel parallel join approach for handling data skew in distributed architectures. We presented the detailed implementation of our method and also evaluated it over a distributed system with different datasets. The experimental results have shown that the new approach has the property of robustness against skew, which can be considered as a supplement for the existing hash-based and duplication-based schemes. Moreover, we have extended our framework to distributed outer joins and proposed the efficient *query with counters* algorithm, which is also shown to be faster than the state-of-the-art method in the presence of high skew.

As the query-based joins performs badly when processing low skewed data, in Chapter 7, we have proposed another more efficient and robust join algorithm referred to as PRPQ (*partial redistribution & partial query*). We have conducted a theoretical analysis and presented detailed implementation as well as evaluations of this algorithm. The experimental results over various join workloads have demonstrated that the PRPQ algorithm is indeed robust and scalable under a wide range of skew conditions. Specifically, compared to the state-of-art PRPD method, we have achieved $16\% - 167\%$ performance improvement and $24\% - 54\%$ less network communication, which confirms our theoretical analysis.

Dynamical Indexing over Distributed Systems

Indexing methods in current distributed RDF systems are not suitable for a high performance data analysis system, as all of them meet performance bottlenecks on either data loading or querying when processing large amounts of data. In Chapter 8, we have proposed an efficient method for processing RDF using dynamic data re-partitioning to enable rapid analysis of large datasets. Our approach adopts a two-tier index architecture on each computation node: (1) a lightweight primary index, to keep loading times low, and (2) a series of dynamic, multi-level secondary indexes, calculated as a by-product of query execution, to decrease or remove inter-machine data movement for subsequent queries that contain the same graph patterns. In addition, we have proposed methods to replace some secondary in-

dexes with distributed filters, so as to decrease memory consumption. Experimental results on a commodity cluster of 16 nodes show that our multi-level indexing approach can indeed improve loading speeds by an order of magnitude while remaining competitive in terms of performance. Our system can load 1.1 billion triples at a rate of 2.48 million triples per second and provides competitive performance to RDF-3X and 4store.

9.2 Future Work

Extension of Current Implementations

The final implementation in Chapter 8 has demonstrated that our designs can very quickly process large scale RDF data. Regardless, we can investigate further extensions to our approach through the application of methods for *skew handling*, *index size reduction* and *incremental sorting* as discussed, which could further improve performance.

Moreover, as our implementation over the distributed system are mainly based on bulk operations, in which the transferred chunks are normally very large. In this scenario, saving network communications would make a great contribution to the performance. Therefore, additional techniques on this aspect can be further investigated. For example, efficient compression/decompression methods for data communication would be very relevant, although there will be a trade-off between the computation and communication cost.

Finally, as we focus on applying parallel techniques to RDF data processing, the given workloads as well as the network topology of test platform in our evaluations is relatively simple. In order to meet the challenges of more complex workloads and networks, efficient scheduling and predictive approaches can be further developed on the basis of our framework, so as to satisfy the latency and throughput requirements. In the meantime, efficient data placement strategies can also be considered, to further exploit locality of data access, and consequently optimize core utilization and reduce network consumption.

Applied to RDF Reasoning

RDF reasoning, namely inferring additional information from the presented RDF data, is a crucial problem for the semantic web. It has been used for deriving higher-level knowledge, assisting decision support and data cleaning. For a RDF management system, it can also be used for enriching the results of queries by adding implicit information.

In terms of big data reasoning, there exist significant challenges, as the research of artificial intelligence traditionally focuses on rich knowledge structures (instead of large scale

data) and the computational cost of many methods central to the area. It is clear that applying parallel techniques on reasoning would also encounter similar computation, network communication and load balancing problems as those solved here.

To be associated with the techniques proposed in this thesis, the dictionary encoding method can be used to improve reasoning ability, as the computation cost can be highly reduced over numbers rather than long strings. Moreover, because the computation of *well-founded semantics* can be modeled as a sequence of join and anti-join operations, it can be foreseen that our proposed join algorithms can also be applied for increasing the reasoning processing performance.

Towards a High Performance RDF Analysis Engine

Currently, our system can compute simple SPARQL queries, however we can further extend the system so as to include a larger set of functionality, such as inclusion of aggregates, and thus be able to run more complex benchmarks. In addition, although our system has much lower coordination overhead than systems based on Hadoop, we can reduce it further by limiting the number of nodes involved in cheap operations.

Moreover, as this thesis is concentrated on system implementations, the details about query plan generation has not been discussed. In fact, similarly to other data management systems, that process will be critical for SPARQL query execution as well. For example, choosing a suitable join order would obviously improve the query performance. In this scenario, efficient strategies as well as detailed time cost model for query plan optimization have to be considered. Additionally, for a completed RDF system, besides the functions of data storage and data querying, other management operations such as data adding/deleting and result materialization need to be considered as well.

We anticipate that with extensions to our framework, a highly efficient distributed management system specifically for analyzing large RDF data can be developed, and this would represent a significant contribution to semantic web applications in this big data era.

9.3 Concluding Remarks

In this thesis, from the basis of studying current triple stores and parallel hashing, we have introduced the design of a scalable framework for analyzing large-scale RDF data. We have presented the details of three core operations, in terms of system implementations: (1) underlying dictionary encoding, (2) parallel joins and (3) indexing operations. We have

proposed various techniques to improve the performance of each of these operations and demonstrated their efficiency through performance comparisons with the state-of-art methods. Moreover, the final system-level evaluation has shown that we can load large RDF data very fast while remaining within an interactive range for query processing.

References

[1] Abadi, D., Madden, S., and Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682.

[2] Abadi, D., Marcus, A., Madden, S., and Hollenbach, K. (2007a). Using the Barton libraries dataset as an RDF benchmark. Technical report.

[3] Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2007b). Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422.

[4] Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2009). SW-Store: A vertically partitioned DBMS for semantic web data management. *The VLDB Journal*, 18(2):385–406.

[5] Adida, B. and Birbeck, M. (2008). RDFa primer: Bridging the human and data webs. http://www.w3.org/TR/xhtml-rdfa-primer/.

[6] Al Hajj Hassan, M. and Bamha, M. (2009). An efficient parallel algorithm for evaluating join queries on heterogeneous distributed systems. In *Proceedings of the 16th annual IEEE International Conference on High Performance Computing*, HiPC '09, pages 350–358.

[7] Albutiu, M.-C., Kemper, A., and Neumann, T. (2012). Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075.

[8] Alcantara, D. A., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J. D., and Amenta, N. (2009). Real-time parallel hashing on the GPU. *ACM Trans. Graph.*, 28(5):154:1–154:9.

[9] Alexander, N., Lopez, X., Ravada, S., Stephens, S., and Wang, J. (2004). RDF data model in Oracle. In *W3C Workshop on Semantic Web for Life Sciences*, volume 186.

[10] Apweiler, R., Bairoch, A., Wu, C. H., Barker, W. C., Boeckmann, B., Ferro, S., Gasteiger, E., Huang, H., Lopez, R., Magrane, M., et al. (2004). UniProt: The universal protein knowledgebase. *Nucleic Acids Research*, 32(suppl 1):D115–D119.

[11] Atre, M., Chaoji, V., Zaki, M. J., and Hendler, J. A. (2010). Matrix "bit" loaded: A scalable lightweight join query processor for RDF data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 41–50.

[12] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. (2007). DBpedia: A nucleus for a web of open data. In *Proceedings of the 6th International Semantic Web Conference*, ISWC '07, pages 722–735.

[13] Berners-Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific American*, 284(5):28–37.

[14] Bhargava, G., Goel, P., and Iyer, B. (1995). Hypergraph based reorderings of outer join queries with complex predicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 304–315.

[15] Bizer, C. and Schultz, A. (2009). The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst*, 5(2):1–24.

[16] Blanas, S., Li, Y., and Patel, J. M. (2011). Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 37–48.

[17] Bornea, M. A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., and Bhattacharjee, B. (2013). Building an efficient RDF store over a relational database. In *Proceedings of the 2013 International Conference on Management of Data*, SIGMOD '13, pages 121–132.

[18] Bröcheler, M., Pugliese, A., and Subrahmanian, V. S. (2009). DOGMA: A disk-oriented graph matching algorithm for RDF databases. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 97–113.

[19] Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying RDF and RDF schema. In *Proceedings of the First Internation Semantic Web Conference*, ISWC '02, pages 54–68.

[20] Cagri Balkesen, Jens Teubner, G. A. and Öszu, M. T. (2013). Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 29th International Conference on Data Engineering*, ICDE '13, pages 362–373.

[21] Chan, C.-Y. and Ioannidis, Y. E. (1998). Bitmap index design and evaluation. In *ACM SIGMOD Record*, volume 27, pages 355–366. ACM.

[22] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538.

[23] Chebotko, A., Lu, S., and Fotouhi, F. (2009). Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering*, 68(10):973–1000.

[24] Chen, Z., Gehrke, J., and Korn, F. (2001). Query optimization in compressed database systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 271–282.

[25] Cheng, L., Kotoulas, S., Ward, T., and Theodoropoulos, G. (2012a). Runtime characterisation of triple stores: An initial investigation. In *Proceedings of the 23rd IET Irish Signals and Systems Conference*, ISSC '12, pages 1–6.

[26] Cheng, L., Kotoulas, S., Ward, T., and Theodoropoulos, G. (2012b). Runtime characterization of triple stores. In *Proceedings of the 15th IEEE International Conference on Computational Science and Engineering*, CSE '12, pages 66–73.

[27] Cheng, L., Kotoulas, S., Ward, T. E., and Theodoropoulos, G. (2013). QbDJ: A novel framework for handling skew in parallel join processing on distributed memory. In *Proceedings of the15th IEEE International Conference on High Performance Computing and Communications*, HPCC '13, pages 1519–1527.

[28] Cheng, L., Kotoulas, S., Ward, T. E., and Theodoropoulos, G. (2014a). Design and evaluation of parallel hashing over large-scale data. In *Proceedings of the 21st IEEE International Conference on High Performance Computing*, HiPC '14.

[29] Cheng, L., Kotoulas, S., Ward, T. E., and Theodoropoulos, G. (2014b). Efficiently handling skew in outer joins on distributed systems. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '14, pages 295–304.

[30] Cheng, L., Kotoulas, S., Ward, T. E., and Theodoropoulos, G. (2014c). RDF-ReHashed: Fast distributed loading and querying of large RDF datasets. In *submission*.

[31] Cheng, L., Kotoulas, S., Ward, T. E., and Theodoropoulos, G. (2014d). Robust and efficient large-large table outer joins on distributed infrastructures. In *Proceedings of the 20th European Conference on Parallel Processing*, Euro-Par '14, pages 258–369.

[32] Cheng, L., Kotoulas, S., Ward, T. E., and Theodoropoulos, G. (2014e). Robust skew-resistant parallel joins in shared-nothing systems. In *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management*, CIKM '14.

[33] Cheng, L., Kotoulas, S., Ward, T. E., and Theodoropoulos, G. (2014f). A two-tier index architecture for fast processing large RDF data over distributed memory. In *Proceedings of the 25th ACM Conference on Hypertext and Social Media*, HT '14.

[34] Cheng, L., Malik, A., Kotoulas, S., Ward, T. E., and Theodoropoulos, G. (2014g). Efficient parallel dictionary encoding for RDF data. In *Proceedings of the 17th International Workshop on the Web and Databases*, WebDB '14.

[35] Chong, E. I., Das, S., Eadon, G., and Srinivasan, J. (2005). An efficient SQL-based RDF querying scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1216–1227.

[36] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to algorithms*. MIT press.

[37] Cyganiak, R. and Jentzsch, A. (2011). Linking open data cloud diagram. http://lod-cloud.net/.

[38] Das, S., Agrawal, D., and El Abbadi, A. (2010). G-store: A scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SCC '10, pages 163–174.

[39] DeWitt, D. and Gray, J. (1992). Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98.

[40] DeWitt, D. J., Naughton, J. F., Schneider, D. A., and Seshadri, S. (1992). Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 27–40.

[41] Dice, D., Hendler, D., and Mirsky, I. (2013). Lightweight contention management for efficient compare-and-swap operations. In *Proceedings of the 19th European Conference on Parallel Processing*, Euro-Par '13, pages 595–606.

[42] Duan, S., Kementsietsidis, A., Srinivas, K., and Udrea, O. (2011). Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 145–156.

[43] El-Ghazawi, T. and Cantonnet, F. (2002). UPC performance and potential: A NPB experimental study. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–26.

[44] Erling, O. and Mikhailov, I. (2010). Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*, pages 501–519.

[45] Fernández, J. D., Gutierrez, C., and Martínez-Prieto, M. A. (2010). RDF compression: basic approaches. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1091–1092.

[46] Frey, P. W., Goncalves, R., Kersten, M., and Teubner, J. (2009). Spinning relations: High-speed networks for distributed join processing. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, pages 27–33.

[47] Galindo-Legaria, C. and Rosenthal, A. (1997). Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems (TODS)*, 22(1):43–74.

[48] García, I., Lefebvre, S., Hornus, S., and Lasram, A. (2011). Coherent parallel hashing. *ACM Trans. Graph.*, 30(6):161:1–161:8.

[49] Glavic, B. and Alonso, G. (2009). Perm: Processing provenance and data on the same data model through query rewriting. In *In proceedings of the 25th IEEE International Conference on Data Engineering*, ICDE '09, pages 174–185.

[50] Goodman, E., Haglin, D., Scherrer, C., Chavarria-Miranda, D., Mogill, J., and Feo, J. (2010). Hashing strategies for the Cray XMT. In *IPDPS Workshop*, pages 1–8.

[51] Goodman, E., Lemaster, M. N., and Jimenez, E. (2011). Scalable hashing for shared memory supercomputers. In *Proceedings of 2011 IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 41:1–41:11.

[52] Groppe, S. (2011). *Data Management and Query Processing in Semantic Web Databases*. Springer.

[53] Guo, Y., Pan, Z., and Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182.

[54] Harris, S., Lamb, N., and Shadbolt, N. (2009). 4store: The design and implementation of a clustered RDF store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, SSWS '09, pages 94–109.

[55] Harris, S. and Shadbolt, N. (2005). SPARQL query processing with conventional relational database systems. In *Web Information Systems Engineering – WISE 2005 Workshops*, pages 235–244.

[56] Harth, A., Umbrich, J., Hogan, A., and Decker, S. (2007). YARS2: A federated repository for querying graph structured data from the web. In *Proceedings of the 6th International Semantic Web Conference*, ISWC '07, pages 211–224.

[57] He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. (2008). Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524.

[58] Herlihy, M. (1991). Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149.

[59] Herlihy, M. and Shavit, N. (2011). *The Art of Multiprocessor Programming*. Elsevier Science.

[60] Hill, G. and Ross, A. (2009). Reducing outer joins. *The VLDB Journal*, 18(3):599–610.

[61] Huang, J., Abadi, D. J., and Ren, K. (2011). Scalable SPARQL querying of large RDF graphs. *Proc. VLDB Endow.*, 4(11):1123–1134.

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

[63] Husain, M. F., Doshi, P., Khan, L., and Thuraisingham, B. (2009). Storage and retrieval of large RDF graph using Hadoop and MapReduce. In *Cloud Computing*, pages 680–686.

[64] Idreos, S., Kersten, M. L., and Manegold, S. (2007). Database cracking. In *CIDR*, pages 68–78.

[65] Idreos, S., Kersten, M. L., and Manegold, S. (2009). Self-organizing tuple reconstruction in column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 297–308.

[66] Imasaki, K. and Dandamudi, S. P. (2002). An adaptive hash join algorithm on a network of workstations. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, page 61.

[67] Ivanova, M. G., Kersten, M. L., Nes, N. J., and Gonçalves, R. A. (2010). An architecture for recycling intermediates in a column-store. *ACM Transactions on Database Systems (TODS)*, 35(4):24.

[68] Ives, Z. G. and Taylor, N. E. (2008). Sideways information passing for push-style query processing. In *Proceedings of the 24th IEEE International Conference on Data Engineering*, ICDE '08, pages 774–783.

[69] Iyer, B. R. and Wilhite, D. (1994). Data compression support in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 695–704.

[70] J. Hammer, M. S. (2001). *Data Structures for Databases*, volume 60.

[71] Kaldewey, T., Lohman, G., Mueller, R., and Volk, P. (2012). GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 55–62.

[72] Karlin, A. R. and Upfal, E. (1986). Parallel hashing: An efficient implementation of shared memory. In *Proceedings of the 18th annual ACM Symposium on Theory of Computing*, STOC '86, pages 160–168.

[73] Karypis, G. and Kumar, V. (1995). Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0.

[74] Khare, R. and Çelik, T. (2006). Microformats: a pragmatic path to the semantic web. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 865–866.

[75] Kim, C., Kaldewey, T., Lee, V. W., Sedlar, E., Nguyen, A. D., Satish, N., Chhugani, J., Di Blas, A., and Dubey, P. (2009). Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow.*, 2(2):1378–1389.

[76] Kim, K., Moon, B., and Kim, H.-J. (2013). R3F: RDF triple filtering method for efficient SPARQL query processing. *World Wide Web*, pages 1–41.

[77] Kossmann, D. (2000). The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469.

[78] Kotoulas, S., Oren, E., and Van Harmelen, F. (2010). Mind the data skew: Distributed inferencing by speeddating in elastic regions. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 531–540.

[79] Kotoulas, S., Urbani, J., Boncz, P., and Mika, P. (2012). Robust runtime optimization and skew-resistant execution of analytical SPARQL queries on Pig. In *Proceedings of the 11th International Semantic Web Conference*, ISWC '12, pages 247–262.

[80] Larson, P.-A. and Zhou, J. (2007). Efficient maintenance of materialized outer-join views. In *In proceedings of the 23rd IEEE International Conference on Data Engineering*, ICDE '07, pages 56–65.

[81] Liu, B. and Hu, B. (2005). An evaluation of RDF storage systems for large data applications. In *SKG*, page 59.

[82] Ma, L., Wang, C., Lu, J., Cao, F., Pan, Y., and Yu, Y. (2008). Effective and efficient semantic web data management over DB2. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1183–1194.

[83] Malard, J. M. and Stewart, R. D. (2002). Distributed dynamic hash tables using IBM LAPI. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–11.

[84] Matias, Y. and Vishkin, U. (1991). On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573 – 606.

[85] Maynard, C. (2011). Comparing UPC and one-sided MPI: A distributed hash table for gap. In *Proceedings of the 5th Conference on Partitioned Global Address Space Programming Models*, PGAS '11.

[86] McBride, B. (2001). Jena: Implementing the RDF model and syntax specification. In *SemWeb*.

[87] McGuinness, D. L., Van Harmelen, F., et al. (2004). OWL web ontology language overview. *W3C recommendation*, 10(2004-03):10.

[88] Morsey, M., Lehmann, J., Auer, S., and Ngomo, A.-C. N. (2011). DBpedia SPARQL benchmark: Performance assessment with real queries on real data. In *Proceedings of the 10th International Semantic Web Conference*, ISWC '11, pages 454–469.

[89] Neumann, T. and Weikum, G. (2008). RDF-3X: A risc-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659.

[90] Neumann, T. and Weikum, G. (2009). Scalable join processing on very large RDF graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 627–640.

[91] Neumann, T. and Weikum, G. (2010). The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113.

[92] Owens, A., Seaborne, A., Gibbins, N., et al. (2008). Clustered TDB: A clustered triple store for Jena.

[93] Pérez, J., Arenas, M., and Gutierrez, C. (2006). Semantics and complexity of SPARQL. In *Proceedings of the 5th Internation Semantic Web Conference*, ISWC '06, pages 30–43.

[94] Raimond, Y., Scott, T., Sinclair, P., Miller, L., Betts, S., and McNamara, F. (2012). Case study: Use of semantic web technologies on the BBC web sites.

[95] Rao, J., Pirahesh, H., and Zuzarte, C. (2004). Canonical abstraction for outerjoin optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 671–682.

[96] Rohloff, K., Dean, M., Emmons, I., Ryder, D., and Sumner, J. (2007). An evaluation of triple-store technologies for large data stores. In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, pages 1105–1114. Springer.

[97] Rohloff, K. and Schantz, R. E. (2010). High-performance, massively scalable distributed systems using the MapReduce software framework: The SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*.

[98] Roussev, B. and Wu, J. (2006). Distributed computing using Java: A comparison of two server designs. *J. Syst. Archit.*, 52(7):432–440.

[99] Sakr, S. and Al-Naymat, G. (2010). Relational processing of RDF queries: A survey. *ACM SIGMOD Record*, 38(4):23–28.

[100] Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., and Pinkel, C. (2008). An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario. In *Proceedings of the 7th International Semantic Web Conference*, ISWC '08, pages 82–97.

[101] Schneider, D. A. and DeWitt, D. J. (1989). A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 110–121.

[102] Shadbolt, N., Hall, W., and Berners-Lee, T. (2006). The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101.

[103] Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., and Manegold, S. (2008). Column-store support for RDF data management: not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563.

[104] Stadler, C., Lehmann, J., Höffner, K., and Auer, S. (2011). LinkedGeoData: A core for a web of spatial open data. *Semantic Web Journal*.

[105] Steinbrunn, M., Peithner, K., Moerkotte, G., and Kemper, A. (1995). Bypassing joins in disjunctive queries. In *VLDB*, volume 95, pages 11–15.

[106] Stivala, A., Stuckey, P. J., Garcia de la Banda, M., Hermenegildo, M., and Wirth, A. (2010). Lock-free parallel dynamic programming. *J. Parallel Distrib. Comput.*, 70(8):839–848.

[107] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. (2003). Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32.

[108] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al. (2005). C-store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564.

[109] Suchanek, F. M., Kasneci, G., and Weikum, G. (2007). Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 697–706.

[110] Sun, Z., Wang, H., Wang, H., Shao, B., and Li, J. (2012). Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799.

[111] Tachmazidis, I., Cheng, L., Kotoulas, S., Antoniou, G., and Ward, T. E. (2014). Massively parallel reasoning under the well-founded semantics using X10. In *submission*.

[112] Thompson, B. and Personick, M. (2009). Bigdata: The semantic web on an open source cloud. In *International Semantic Web Conference*.

[113] Udrea, O., Pugliese, A., and Subrahmanian, V. (2007). GRIN: A graph based RDF index. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1465.

[114] Umbrich, J., Karnstedt, M., Hogan, A., and Parreira, J. X. (2012a). Freshening up while staying fast: Towards hybrid SPARQL queries. In *Knowledge Engineering and Knowledge Management*, pages 164–174.

[115] Umbrich, J., Karnstedt, M., Hogan, A., and Parreira, J. X. (2012b). Hybrid SPARQL queries: fresh vs. fast results. In *Proceedings of the 11th International Semantic Web Conference*, ISWC '12, pages 608–624.

[116] Urbani, J., Maassen, J., Drost, N., Seinstra, F., and Bal, H. (2013). Scalable RDF data compression with MapReduce. *Concurrency and Computation: Practice and Experience*, 25(1):24–39.

[117] W3C. Resource description framework. http://www.w3.org/RDF/.

[118] W3C. SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

[119] Walton, C. B., Dale, A. G., and Jenevein, R. M. (1991). A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 537–548.

[120] Weaver, J. and Williams, G. T. (2009). Scalable RDF query processing on clusters and supercomputers. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, SSWS '09.

[121] Weiss, C., Karras, P., and Bernstein, A. (2008). Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019.

[122] Westmann, T., Kossmann, D., Helmer, S., and Moerkotte, G. (2000). The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67.

[123] Wick, M. (2008). Geonames.

[124] Wong, K.-F., Yu, J. X., and Tang, N. (2006). Answering XML queries using path-based indexes: A survey. *World Wide Web*, 9(3):277–299.

[125] Wu, S., Jiang, S., Ooi, B. C., and Tan, K.-L. (2009). Distributed online aggregations. *Proc. VLDB Endow.*, 2(1):443–454.

[126] Xu, Y. and Kostamaa, P. (2010). A new algorithm for small-large table outer joins in parallel DBMS. In *Proceedings of the 26th IEEE International Conference on Data Engineering*, ICDE '10, pages 1018–1024.

[127] Xu, Y., Kostamaa, P., Zhou, X., and Chen, L. (2008). Handling data skew in parallel joins in shared-nothing systems. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1043–1052.

[128] Yang, S., Yan, X., Zong, B., and Khan, A. (2012). Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 517–528.

[129] Zeng, K., Yang, J., Wang, H., Shao, B., and Wang, Z. (2013). A distributed graph engine for web scale RDF data. In *Proceedings of the 39th international conference on Very Large Data Bases*, VLDB '13, pages 265–276.

[130] Zhang, C., Xie, C., Xiao, Z., and Chen, H. (2011). Evaluating the performance and scalability of MapReduce applications on X10. In *Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies*, APPT '11, pages 46–57.

[131] Zhang, D. and Larson, P.-A. (2012). Lhlf: lock-free linear hashing. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 307–308.

[132] Zhang, X., Kurc, T., Pan, T., Catalyurek, U., Narayanan, S., Wyckoff, P., and Saltz, J. (2004). Strategies for using additional resources in parallel hash-based join algorithms. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, HPDC '04, pages 4–13.

[133] Zhou, X. and Orlowska, M. E. (1995). Handling data skew in parallel hash join computation using two-phase scheduling. In *Proceedings of The 1st International Conference on Algorithms and Architectures for Parallel Processing*, ICAPP '95, pages 527–536.

Appendix A

The Detailed Implementation of *Query with Counters*

R Distribution

```
1: finish async at p ∈ P {
2: Initialize R_c:array[array[tuple]](N)
3: for tuple ∈ list_of_R do
4:     des ← hash(tuple.key)
5:     R_c(des).add(tuple)
6: end for
7: for i ← 0..(N − 1) do
8:     Serialize R_c(i) to ser_R_c(i)
9:     Push ser_R_c(i) to r_R_c(i)(here) at place i
10: end for
11: }
```

Push Query Keys

```
12: finish async at p ∈ P {
13: Initialize T:array[hashmap[key,ArrayList(value)]](N)
14: for tuple ∈ list_of_S do
15:     des ← hash(tuple.key);
16:     if tuple.key ∉ T(des) then
17:         T(des).put(tuple.key,tuple.value)
18:     else
19:         T(des).get(tuple.key).value.add(tuple.value)
```

```
20:     end if
21: end for
22: for i ← 0..(N − 1) do
23:     Extract keys in T(i) to local_key_c(here)(i)
24:     Serialize local_key_c(here)(i) to ser_key(i)
25:     Push ser_key(i) to remote_key_c(i)(here) at place i
26: end for
27: }
```

Count Matches and Return Queried Values

```
28: finish async at p ∈ P {
29: Initialize T′:hashmap, value_c:array[value]
30: for i ← 0..(N − 1) do
31:     Deserialize r_R_c(here)(i) to tuples
32:     Put all <tuple.key, (tuple.value, 0)> into T′
33: end for
34: for i ← 0..(N − 1) do
35:     Deserialize remote_key_c(here)(i) to key_c
36:     for key ∈ key_c do
37:         if key ∈ T′ then
38:             value_c.add(T′.get(key).value)
39:             T′.get(key).counter++
40:         else
41:             value_c.add(null)
42:         end if
43:     end for
44:     Push value_c(i) to r_value_c(i)(here) at place i
45: end for
46: }
```

Results Lookups State `finish async at` $p \in P$ {

```
47: for i ← 0..(N − 1) do
48:     Deserialize r_value_c(here)(i) to local_value_c
49:     for value ∈ local_value_c do
50:         if value ≠ null then
```

```
51:             Look corresponding key in T(i)
52:             Output matched results
53:         end if
54:     end for
55: end for
56: for key ∈ T′ do
57:     if T′.get(key).counter == 0 then
58:         Output non-matched results
59:     end if
60: end for
61: }
```

Appendix B

Rewritten LUBM SPARQL Queries

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

Q1:

```
select    ?x
where {   ?x rdf:type ub:GraduateStudent.
          ?x ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0>. }
```

Q2:

```
select    ?x  ?y  ?z
where {   ?x rdf:type lubm:GraduateStudent.
          ?y rdf:type lubm:Department.
          ?z rdf:type lubm:University.
          ?y lubm:subOrganizationOf ?z.
          ?x lubm:memberOf ?y.
          ?x lubm:undergraduateDegreeFrom ?z. }
```

Q3:

```
select    ?x
where {   ?x rdf:type ub:Publication.
          ?x ub:pub-licationAuthor <http://www.Department0.University0.edu/AssistantProfessor0>.}
```

Q4:

```
select    ?x  ?y1  ?y2  ?y3
where {   ?x rdf:type ub:FullProfessor.
          ?x ub:worksFor <http://www.Department0.University0.edu>.
          ?x ub:name ?y1.
          ?x ub:emailAddress ?y2.
          ?x ub:telephone ?y3.}
```

Q5:

```
select    ?x
where {   ?x rdf:type ub:GraduateStudent.
          ?x ub:memberOf <http://www.Department0.University0.edu>.}
```

Q6:

```
select   ?x
where {  ?x rdf:type ub:GraduateStudent.}
```

Q7:

```
select   ?x ?y
where {  ?x rdf:type ub:GraduateStudent.
         ?y rdf:type ub:Course.
         ?x ub:takesCourse ?y.
         <http://www.De-partment0.University0.edu/AssociateProfessor0> ub:teacherOf ?y.}
```

Q8:

```
select   ?x ?y ?z
where {  ?x rdf:type ub:GraduateStudent.
         ?y rdf:type ub:Department.
         ?x ub:memberOf ?y.
         ?y ub:sub-OrganizationOf <http://www.University0.edu>.
         ?x ub:em-ailAddress ?z.}
```

Q9:

```
select   ?x ?y ?z
where {  ?x rdf:type ub:GraduateStudent.
         ?y rdf:type ub:FullProfessor.
         ?z rdf:type ub:Course.
         ?x ub:advisor ?y.
         ?y ub:teacherOf ?z.
         ?x ub:takesCourse ?z.}
```

Q10:

```
select   ?x
where {  ?x rdf:type ub:GraduateStudent.
         ?x ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0>.}
```

Q11:

```
select   ?x
where {  ?x rdf:type ub:ResearchGroup.
         ?x ub:subOrganizationOf <http://www.University0.edu>.}
```

Q12:

```
select   ?x ?y
where {  ?x rdf:type ub:FullProfessor.
         ?y rdf:type ub:Department.
         ?x ub:worksFor ?y.
         ?y ub:subOrganizationOf <http://www.University0.edu>.}
```

Q13:

```
select   ?x
where {  ?x rdf:type ub:GraduateStudent.
         ?x ub:undergraduateDegreeFrom <http://www.University0.edu>.}
```

Q14:

```
select   ?x
where {  ?x rdf:type ub:UndergraduateStudent.}
```