

Sven Groppe

Data Management and Query Processing in Semantic Web Databases

 Springer

Data Management and Query Processing in Semantic Web Databases

Sven Groppe

Data Management and Query Processing in Semantic Web Databases

 Springer

Sven Groppe
Institute of Information Systems
University of Lübeck
Ratzeburger Allee 160 (Building 64 - 2nd level)
23562 Lübeck
Germany
groppe@ifis.uni-luebeck.de

ISBN 978-3-642-19356-9 e-ISBN 978-3-642-19357-6
DOI 10.1007/978-3-642-19357-6
Springer Heidelberg Dordrecht London New York

ACM Computing Classification (1998): H.2, H.3, I.2

Library of Congress Control Number: 2011926984

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover design: deblik

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

1	Introduction	1
1.1	Main Target Group of the Book	2
1.2	Prerequisites Needed to Understand the Book	3
1.3	Content	3
1.4	Logical Organization of the Book	4
1.5	Structure of the Chapters and Book Webpage	4
2	Semantic Web	7
2.1	Introduction	7
2.2	Overview	8
2.3	RDF Data	9
2.3.1	N3 Notation	11
2.3.2	RDF/XML	13
2.4	Ontology Languages	13
2.5	Open World Assumption	16
2.6	No Unique Name Assumption	17
2.7	SPARQL Query Language	17
2.7.1	Language Constructs of SPARQL	18
2.7.2	SPARQL Protocol for RDF	24
2.7.3	SPARQL Query Results XML Format	26
2.7.4	RDF Stores	27
2.8	Rules	28
2.9	Related Work	31
2.9.1	RIF Processing	31
2.9.2	Optimizations for Recursive Rules	33
2.10	Summary and Conclusions	34
3	External Sorting and B⁺-Trees	35
3.1	Motivation	35
3.2	B ⁺ -trees	36
3.2.1	Properties of B ⁺ -Trees	37

3.2.2	Self-balancing Property of B ⁺ -Trees	38
3.2.3	Searching	39
3.2.4	Prefix Search in Combination with Sideways Information Passing	39
3.2.5	Inserting	41
3.2.6	Deleting	43
3.2.7	B ⁺ -Tree Construction from a large Dataset	45
3.3	Heap	45
3.4	(External) Merge Sort	47
3.5	Replacement Selection	48
3.6	External Chunks Merge Sort	50
3.7	Distribution Sort	52
3.8	RDF Distribution Sort	53
3.9	Experimental Analysis	56
3.9.1	SP ² B Dataset	57
3.9.2	Yago Dataset	58
3.10	Summary and Conclusions	63
4	Query Processing Overview	67
4.1	The LUPOSDATE System	67
4.2	Phases of Query Processing	69
4.3	CoreSPARQL	73
4.3.1	Defining CoreSPARQL	73
4.3.2	Transforming SPARQL Queries into CoreSPARQL Queries	74
4.3.3	CoreSPARQL Grammar	77
4.4	Related Work	78
4.5	Summary and Conclusions	78
5	Logical Optimization	79
5.1	Logical Algebra	79
5.1.1	Semantics of the Logical Algebra Operators	81
5.2	Logical Optimization Rules	85
5.2.1	Pushing FILTER Operators	85
5.2.2	Splitting and Commutativity of FILTER Operators	87
5.2.3	Constant and Variable Propagation	87
5.2.4	Heuristic Query Optimization Using Equivalency Rules	89
5.2.5	Cost-Based Optimization	90
5.2.6	Histograms	99
5.3	Further Related Work	101
5.4	Summary and Conclusions	101
6	Physical Optimization	103
6.1	Motivation	104
6.2	Related Work	106

- 6.3 Indexing 108
 - 6.3.1 Building In-Memory Indices 109
 - 6.3.2 Building Disk-Based Indices 110
- 6.4 Pipelining Versus Materialization 116
 - 6.4.1 Pipeline-Breaker 116
 - 6.4.2 Sideways Information Passing 116
- 6.5 Join Algorithms 117
 - 6.5.1 Nested-Loop Join 117
 - 6.5.2 Merge Join 120
 - 6.5.3 Index Join 122
 - 6.5.4 Hash Join 123
- 6.6 Dynamically Restricting Triple Patterns 126
- 6.7 Sorting Numbering Scheme 129
 - 6.7.1 Joins Without Presorting Numbers 129
 - 6.7.2 Joins with Presorting Numbers 131
 - 6.7.3 Optimization of Fast Sorting 132
 - 6.7.4 Sorting for Complex Joins 132
 - 6.7.5 Additional Benefits from SIP Strategies 135
- 6.8 Optional 136
 - 6.8.1 MergeOptional 136
- 6.9 Duplicate Elimination 137
 - 6.9.1 Duplicate Elimination Using Hashing 137
 - 6.9.2 Duplicate Elimination Using Sorting 138
 - 6.9.3 Duplicate Elimination Using Presorting Numbers 138
- 6.10 Cost Model 138
- 6.11 Performance Evaluation 139
 - 6.11.1 Performance Evaluation for In-memory Databases 139
 - 6.11.2 Performance Evaluation for Large-Scale Datasets 145
- 6.12 Summary and Conclusions 152

- 7 Streams 155**
 - 7.1 Introduction 155
 - 7.2 eBay 156
 - 7.3 Monitoring eBay Auctions 157
 - 7.3.1 Monitoring System 157
 - 7.3.2 Demonstration 158
 - 7.3.3 Streaming SPARQL Engine 159
 - 7.4 Special Operators for Stream Processing 160
 - 7.4.1 Types of Stream Operators 160
 - 7.4.2 Types of Window Operators 161
 - 7.5 Related Work 161
 - 7.5.1 Data Streams in General 161
 - 7.5.2 Semantic Web Data Streams 162
 - 7.6 Summary and Conclusions 162

8	Parallel Databases	163
8.1	Motivation	163
8.2	Types of Parallelisms	165
8.3	Amdahl's Law	167
8.4	Parallel Monitors and Bounded Buffers	168
8.5	Parallel Join Using a Distribution Thread	168
8.6	Parallel Merge Join Using Partitioned Input	169
8.7	Parallel Computation of Operands	172
8.8	Performance Evaluation	173
8.9	Performance Gains and Loss	175
8.10	Summary and Conclusions	175
9	Inference	177
9.1	Introduction	177
9.2	RDF Schema Inference Rules	178
9.3	Materialization of Inference and Consequences for Query Optimization	179
9.4	Logical Optimization for Inference	180
9.5	Performance Analysis	187
9.6	Related Work	189
9.7	Summary and Conclusions	189
10	Visual Query Languages	191
10.1	Motivation	191
10.2	Related Work	193
10.3	RDF Visual Editor	194
10.4	SPARQL Visual Editor	194
10.5	Browser-Like Query Creation	194
10.6	Generating Condensed Data View	196
10.7	Refining Queries	197
10.8	Query Formulation Demo	198
10.9	Computation of Suggested Triple Patterns for Query Refinement	199
10.10	Summary and Conclusions	201
11	Embedded Languages	203
11.1	Motivation	203
11.2	Related Work	204
11.3	Embedding Semantic Web Languages Into JAVA	205
11.3.1	The Type System	208
11.3.2	Subtype Test	210
11.3.3	Satisfiability Test of Embedded SPARQL and SPARUL Queries	215
11.3.4	Determination of the Query Result Types	217
11.4	Summary and Conclusions	217

- 12 Comparison of the XML and Semantic Web Worlds** 219
 - 12.1 Introduction 219
 - 12.2 Concepts and Visions 221
 - 12.3 Data Models 221
 - 12.4 Schema and Ontology Languages 222
 - 12.5 Query Languages 223
 - 12.6 Embedding SPARQL into XQuery/XSLT 226
 - 12.6.1 Embedded SPARQL 226
 - 12.6.2 Translation Process 229
 - 12.6.3 Experimental Analysis 235
 - 12.7 Embedding XPath Into SPARQL 240
 - 12.7.1 Translation of XPath Subqueries Into SPARQL
Queries 241
 - 12.7.2 Performance Analysis 247
 - 12.8 Related Work 248
 - 12.9 Summary and Conclusions 250

- 13 Summary, Conclusions, and Future Work** 251
 - 13.1 Possibilities for Future Work 252

- References** 255

- Index** 267

Chapter 1

Introduction

The current World Wide Web (short *Web*) enables an easy, instant access to a vast amount of online information. However, the content in the Web is typically for human consumption and is not tailored to be machine-processed.

Machines already help users to organize their lives in the Internet. The most prominent examples are search engines such as google or yahoo: Users type some keywords as “food” and the search engine displays the webpages containing these keywords. During its search, the search engine compares plain text rather than the meaning of the keywords with the words on a webpage. Consequently, a number of unwanted webpages are displayed to the user, since for example, the meaning of a given keyword is ambiguous or the webpages contain only little relevant content for this keyword. On the other hand, if a meaning is expressed by using other words than the given keyword, relevant webpages are not retrieved by the search engine. If a search engine can understand the meaning of a webpage and can use the meaning during search, the described problems occur less and the quality of search results can be improved.

The Semantic Web, which is intended to establish a machine-understandable web, thereby offers a promising and potential solution to mining and analyzing web content. The advocators of the Semantic Web define the Semantic Web as “an extension of the current web, in which information is given a well-defined meaning, better enabling computers and people to work in cooperation” (Berners-Lee et al. 2001).

Around the vision of the Semantic Web, a number of semantic standards and techniques have been developed by the World Wide Web Consortium (W3C), among which the *Resource Description Framework* (RDF) is an important one. RDF provides a general method for conceptual description and is designated by the Semantic Web as its data model to describe the web resources. The W3C also developed SPARQL as RDF querying language.

The Semantic Web is currently changing from an emergent trend to a technology used in complex real-world applications. Semantic Web ontologies and RDF knowledge bases are becoming increasingly large. The examples of large RDF data with millions and even billions of facts include the UniProt comprehensive

catalogue of protein sequence, function, and annotation data (Swiss Institute of Bioinformatics 2009), the RDF data extracted from Wikipedia (Auer et al. 2007), the Princeton University's WordNet (Assem et al. 2006), and the Billion Triples Challenge (Semantic web challenge 2010). The querying performance is no doubt a key issue for Semantic Web applications.

In this book, we shed light on various aspects of high performance Semantic Web data management and query processing, which is the heart of most applications and is often most worth to be optimized.

Why it is necessary to have a book specialising on the data management and query processing in Semantic Web databases? Why not only read a book about the Semantic Web and another one about databases?

The existing Semantic Web books typically focus on presenting the Semantic Web technologies and their applications. After reading these books, the readers get to know how to use Semantic Web technologies. We have just mentioned that Semantic Web databases are becoming increasingly large, and the Billion Triples Challenge (Semantic web challenge 2010) contains already several billion triples. Semantic Web data managing and querying performance is no doubt a key issue for Semantic Web applications. However, no books examine high performance Semantic Web applications and address how to exploit the Semantic Web properties to speed up the applications of Semantic Web.

On the other hand, database books usually describe the database internals for data management and query processing specialized to the relational world. Although they may cover also topics about object-oriented databases, web databases (without the Semantic Web), XML databases, and other topics, they fail to address the special topics about Semantic Web databases. The Semantic Web databases deal with the special index structures and data management and query processing approaches, which exploit the properties of the Semantic Web data model and the Semantic Web query languages to speed up applications.

This book fills the gap of Semantic Web books and database books and deals especially with how to use Semantic Web data and query specific properties to efficiently manage and query very large Semantic Web databases.

1.1 Main Target Group of the Book

The main target group include as follows:

- Lecturers and students at universities and universities of applied sciences
- Researchers in the area of the Semantic Web and databases
- Software developers for semantic web databases
- Readers, which study on these topics
- All who are interested in databases and the Semantic Web, and especially in query processing and data management of Semantic Web databases

1.2 Prerequisites Needed to Understand the Book

The main prerequisites needed to understand the book are as follows:

- Basic computer science knowledge
- Basic knowledge about algorithms
- Semantic Web knowledge is not a prerequisite as the necessary Semantic Web knowledge will be introduced in this book
- No knowledge about database internals such as query processing and data management is required as those topics will also be introduced in this book

1.3 Content

We start with describing the specifications of the Semantic Web in Chap. 2. All other chapters deal with query processing specialized to the Semantic Web world. Chapter 3 contains an excursion to external sorting and B^+ -trees. B^+ -trees are later used as index for large-scale datasets and external sorting is used, for example, as part for an efficient construction of a B^+ -tree from scratch. We have developed an own general external sorting algorithm with higher performance than traditional external sorting algorithms and a specialized external sorting algorithm for speeding up index construction of large-scale Semantic Web indices. Chapter 4 introduces query processing and its phases. We also introduce our Semantic Web database management system LUPOSDATE and the transformation of Semantic Web queries into a core of the query language without redundant language constructs to simplify further processing steps in this chapter. Chapters 5 and 6 provide more details of the main query processing phases logical and physical optimizations. Besides explaining logical optimization rules, we describe the sophisticated query optimizer in the LUPOSDATE implementation. We have developed indices for large-scale as well as main memory databases for high speed query processing and a sorting numbering scheme for fast sorting in order to avoid costly hash join operations. When all these proposed techniques play together, we can process typical queries on even very large datasets with over one billion triples in seconds or often in milliseconds, thus making Semantic Web applications feasible. Chapter 7 deals with stream processing, that is, the processing of (possibly infinite) streams of data generated, for example, by sensors. Chapter 8 describes the optimization possibilities when using parallel database technologies of today's multicore processors. Chapter 9 describes different optimization possibilities and materialization strategies for supporting inference. Chapter 10 introduces visual query languages as alternative to formulate textual queries. Chapter 11 deals with query and data languages embedded into programming languages as well as their features. Chapter 12 finally compares the Semantic Web world with the XML world as alternative for web data and queries. Chapter 13 provides summary and conclusions as well as hints for future work.

1.4 Logical Organization of the Book

Figure 1.1 presents the logical organization of the book, that is, the dependencies between the chapters. An arrow from a chapter X to a chapter Y means that the content of chapter X is directly precondition of (and used within) the chapter Y. Note that we did not mark all dependencies, but have drawn the main ones; that is, small dependencies that should not hinder the main understanding if the chapter is left away. Although we recommend to read the book from the first to the last chapter, this logical organization may help the reader to read other passes through the book and pick the chapter(s) he is only interested in (and those which are required for these chapters).

1.5 Structure of the Chapters and Book Webpage

Each chapter starts with an abstract. After the abstract, an introduction shortly introduces the topic of the chapter. Each chapter ends up with summary and conclusions. The book webpage at <http://www.ifis.uni-luebeck.de/~groppe/SemWebDBBook/> contains additional material such as exercises for each chapter. The exercises can be used by lecturers as material for tutorials and by interested

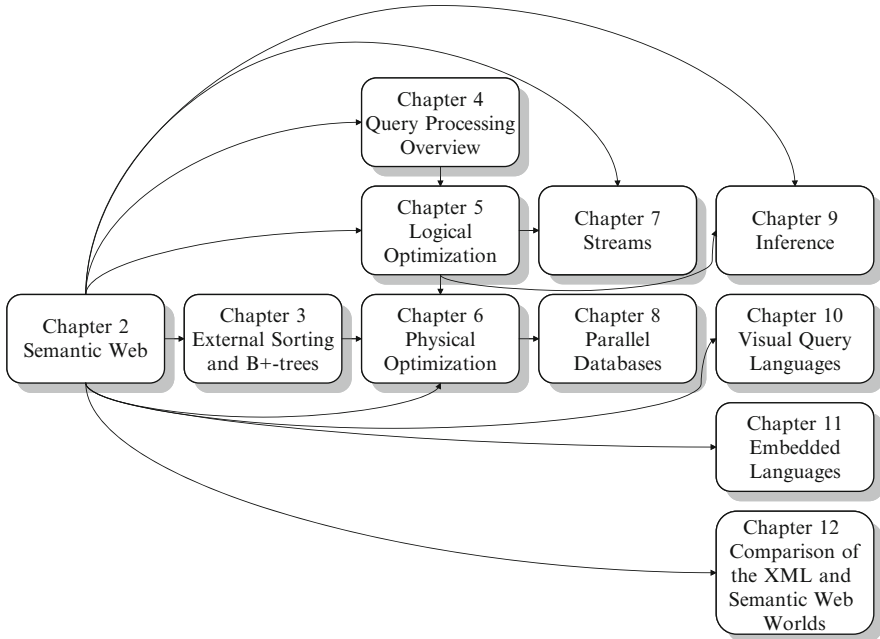


Fig. 1.1 Logical organization of the book

readers to check their understanding and learning success. The solutions of the exercises are also given at the book webpage.

Acknowledgements First of all I thank Jinghua and Nils for the valuable family life they gave to me. I thank for their patience, their humor, exciting moments of life, and last, but not least for their love.

I thank all who have contributed to the implementations of the discussed approaches and all who gave comments and remarks, and who have proposed improvements for the implementations and online demonstrations.

I thank the reviewers of my papers for providing advices and their stimulations to go on with my research after kind words and after acceptance.

I thank the students of my lectures, tutorials and laboratories, and those with a bachelor and master thesis supervised by me for testing and extending the implementations.

I thank the members of the Institute of Information Systems (IFIS) for the smart working environment and the sometimes long, but useful and fruitful discussions.

Chapter 2

Semantic Web

Abstract The Semantic Web provides languages to define data, queries, ontologies, and rules. This chapter introduces them after a short motivation and overview of the Semantic Web.

2.1 Introduction

Although Semantic Web technologies can be and are used independent of the Internet, the design issues of the Semantic Web address challenges for processing data and knowledge in the internet such as vastness, semantically duplicated terms, vagueness, incompleteness, uncertainty, and inconsistency:

- *Vastness*: The indexed web already contains at least 21.13 billion pages as reported in (de Kunder et al. 2010) for the 6 May 2010. As well as the traditional World Wide Web was designed to publish webpages read by humans, Semantic Web technologies such as RDFa (Adida and Birbeck 2008) allow webpage designers to embed RDF in their webpages for encoding information to be processed by machines. Other Semantic Web datasets with up to over one billion facts were crawled from data of the web or built for a special domain like the UniProt comprehensive catalog of protein sequence, function, and annotation data (Swiss Institute of Bioinformatics 2009), the RDF data extracted from Wikipedia (Auer et al. 2007), the Princeton University's WordNet (Assem et al. 2006), and the Billion Triples Challenge (Semantic web challenge 2010). Semantic Web tools have to be able to process these truly large datasets.
- *Semantically duplicated terms*: Large datasets may contain semantically duplicated terms referring to the same thing such as truck, lorry, and freight vehicle, which complicates automated processing and must be dealt with.
- *Vagueness*: Humans understand vague terms such as *young* and can handle different meanings of the same term in different contexts: a young child may have a different understanding of a *young* person as an adult or old person. Furthermore, even in the same context, it is not absolutely clear at what exact age a person is young and at what exact age a person is not young any more.

Much information contains such vague terms, such that machines must be able to process them.

- *Incompleteness*: New data sources occur frequently in the internet; other data sources disappear or are temporarily unavailable. Thus, information in the internet can never be seen as being complete, as new or temporarily unavailable data sources can contribute relevant information for a specific topic. Therefore, Semantic Web tools should regard any available information as being incomplete.
- *Inconsistency*: Due to the huge amount of information in the internet, contradictory information can be given in different or even in the same data sources. Semantic Web tools must react on contradictory information in a reasonable and practical way.

To deal with some of these challenges, Semantic Web technologies consider the meaning of symbols during processing. This reduces the number of errors when, for example, searching for or automatically integrating data and services. For this purpose, Semantic Web technologies also infer new knowledge on the basis of ontologies and rules. Inference helps to avoid redundancies in data and knowledge, as inferred data and knowledge neither need to be explicitly stated nor stored. Furthermore, inference aims to detect hidden relationships within the data and knowledge. However, errors in data and knowledge can lead to unwanted detected relationships and thus unwanted results. Therefore, it is necessary to check carefully the data, knowledge, and inferred data by humans.

After a short overview of the basic architecture of the Semantic Web in the next subsection, we introduce its basic specifications. Note that we cannot cover all specifications, drafts, and planned specifications here, but we focus on those which are relevant for the topics addressed in this book. The reader should keep in mind that a lot more is going on in the quite active Semantic Web community and the World Wide Web is still the best source to be informed about recent activities.

2.2 Overview

Figure 2.1 presents the basic architecture of the Semantic Web, including the important Semantic Web specifications from W3C, and their relationships. The *Resource Description Framework* (RDF) (Beckett 2004), which defines a simple

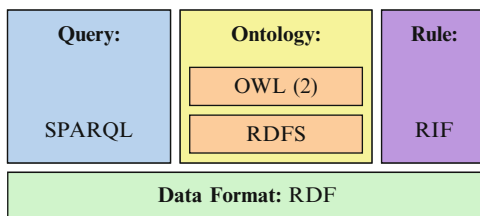


Fig. 2.1 Basic semantic web architecture

and extremely flexible data model, is an underlying component in this architecture. The concepts of ontologies and of rules are the basis for the data integration and inference of new facts. The standard languages for constructing ontologies are *RDF Schema* (RDFS) (Brickley and Guha 2004) and the *Web Ontology Language* (OWL) (Dean and Schreiber 2004) with its successor OWL 2 (Motik et al. 2009). The W3C also proposes the *Rule Interchange Format* (RIF) (Boley et al. 2009) as RDF rule language, and SPARQL (Prud'hommeaux and Seaborne 2008) as RDF query language.

Since RDF and SPARQL are two important technologies dealt with in this book, we will describe them in detail in this chapter. For the other specifications we provide only background information.

2.3 RDF Data

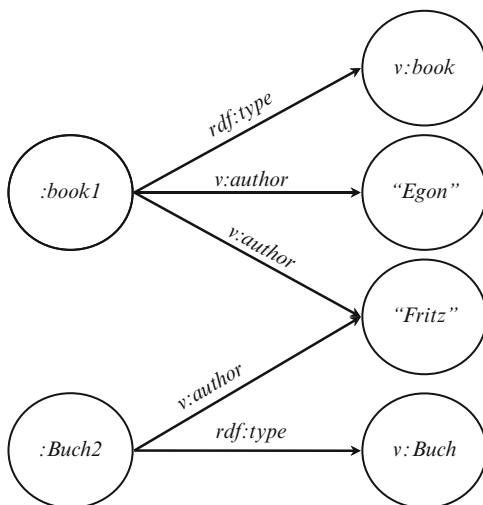
The RDF data model is based upon the idea of making statements about resources (in particular Web resources) and is therefore similar to classic conceptual modeling approaches such as Entity-Relationship or Class diagrams. The statements are subject–predicate–object expressions of the form (*Subject, Predicate, Object*) and are known as *triples* in RDF terminology. *Subject* indicates a resource (also known as entity), *Predicate* represents a property of the entity, and *Object* is a value of the property in form of a resource or literal. The predicate is also often explained as expressing a relationship between the subject and object. For example, one way to represent the notion “book1 has the author Egon” in RDF is as the triple (s, p, o), where the subject s denotes “book1”, the predicate p “has the author”, and the object o “Egon”.

RDF data consist of a set of triples. The set of triples builds a directed graph, called *RDF graph*, where the subject and object are nodes and the predicate is a labeled directed edge from the subject to the object. Nodes are unique; that is, the same resources and literals in subjects and objects are represented by the same node. For an example of an RDF graph, see Fig. 2.2.

RDF is an abstract model with several serialization formats (i.e., file formats) such as RDF triples (Grant and Beckett 2004), N3 (Berners-Lee 1998), Turtle (Beckett 2006), or RDF/XML (Beckett 2004), which uses XML to encode RDF data. In the following subsections, we shortly introduce the N3 notation and RDF/XML, which are the most widely used formats due to the human-readable syntax of N3 and the integration of RDF/XML into the RDF specification.

Resources are described in RDF by Internationalized Resource Identifiers (IRIs) (Dürst and Suignard 2005). An IRI is a complement to the Uniform Resource Identifier (URI) (Berners-Lee et al. 1998). A mapping from IRIs to URIs is defined; that is, IRIs can be used instead of URIs, where appropriate, to identify resources. The syntax of IRIs differs only slightly to the one of URIs. An example of an IRI is <http://www.example.org>. Note that the IRI does not necessarily refer to a real webpage.

Fig. 2.2 Example of an RDF graph



IRIs in RDF notations such as N3 and in query languages such as SPARQL are often enclosed with \langle and \rangle , for example, $\langle \text{http://www.example.org} \rangle$.

Sometimes the user does not want or does not need to provide an explicit IRI as global identifier for a resource. For this purpose, the user can define blank nodes. If available, labels of blank nodes are *local*; that is, local labels can be used within the same RDF file to refer to the same blank nodes, but cannot be used to externally address this blank node. In N3 notation, every $[\]$ defines a new blank node. $_:b$ defines a blank node with local blank node label b .

Literals are values such as strings, numerical values, language-tagged values, or values with a user-defined data type. “*text*” defines the simple literal *text* in N3 notation and in SPARQL.

Typed literals are literals with the further information of its data type. For example, “*text*” $^{\wedge}\langle \text{http://www.w3.org/2001/XMLSchema\#string} \rangle$ is a typed literal, where “*text*” is typed with the XML Schema data type string (Peterson et al. 2009) and which is equivalent to the simple literal “*text*”. Using a prefix, for example, *xsd*, for $\text{http://www.w3.org/2001/XMLSchema\#}$, the typed literal can be abbreviated as “*text*” $^{\wedge}\text{xsd:string}$.

Numerical values are typically expressed by a typed literal with a numerical XML Schema data type (Peterson et al. 2009) such as int, integer, positiveInteger, long, double, float, and decimal. Note that the validation of typed literals is not part of RDF itself and left to the application using RDF. For example, “*not an int*” $^{\wedge}\text{xsd:int}$ is correct in RDF, but applications may expect a real XML Schema *int* value like “120” $^{\wedge}\text{xsd:int}$.

An own user-defined data type may be used within a typed literal, for example, “*content*” $^{\wedge}\langle \text{http://www.myDatatypes.org/myDatatype1} \rangle$. Again, it is up to the application to validate and interpret the content *content* according to the data type $\langle \text{http://www.myDatatypes.org/myDatatype1} \rangle$. In multilingual environments, language-tagged literals of the form “*content*” $@\text{tag}$ may be used, where *tag*

conforms to Alvestrand et al. (2001); for example, language-tagged literals for the English content “hello”@en and for German content “hallo”@de may be used.

For later usage, we define the terms RDF triple and graph in a formal way:

Definition 1 (RDF triple). Assume there are pairwise disjoint infinite sets I , B , and L , where I represents the set of IRIs (Dürst and Suignard 2005), B the set of blank nodes, and L the set of literals. We call a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ an RDF triple, where s represents the subject, p the predicate, and o the object of the RDF triple.

Furthermore, we call an element $e \in I \cup B \cup L$ an RDF term.

Definition 2 (RDF graph). An RDF graph (Beckett 2004) is a set of RDF triples.

2.3.1 N3 Notation

Figure 2.3 contains an example of RDF data in N3 notation, which is the serialization of the RDF graph in Fig. 2.2. The serialized RDF data contain three prefix declarations [see lines (1)–(3)] and a set of triples [see lines (4)–(8)]. A prefix declaration **@prefix name:** <iri> defines an alias *name* for a prefix IRI (Dürst and Suignard 2005) *iri*. The prefix name can be used in the declaration of triples, where *name:postfix* represents the IRI <iri postfix>. For example, *rdf:type* in line (4) represents <<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>> according to the prefix declaration in line (3). The first prefix declaration in line (1) does not define a name for its prefix and is thus a default prefix to be used for prefixed IRIs without prefix name like *:book1* [line (4)] which represents <<http://book/instances/book1>>.

There are some abbreviations in N3 notation.

Object lists separate objects by a comma, and can be used to avoid repeating the same subject and predicate, for example, *:book1 v:author “Fritz”, “Egon”*. abbreviates the lines (5) and (6) of Fig. 2.3. Using predicate–object lists to separate lists of predicate and object by a semicolon avoids repeating the same subject, for example, *:book1 rdf:type v:book; v:author “Fritz”*. is equivalent to the lines (4) and (5) of Fig. 2.3. The N3 abbreviations can usually be arbitrarily mixed; for example,

- (1) **@prefix:** <<http://book/instances/>>.
- (2) **@prefix v:** <<http://book/vocabulary/>>.
- (3) **@prefix rdf:** <<http://www.w3.org/1999/02/22-rdf-syntax-ns#>>.
- (4) *:book1 rdf:type v:book*.
- (5) *:book1 v:author “Fritz”*.
- (6) *:book1 v:author “Egon”*.
- (7) *:Buch2 rdf:type v:Buch*.
- (8) *:Buch2 v:author “Fritz”*.

Fig. 2.3 RDF data in N3 Notation

the triples in the lines (4)–(6) of Fig. 2.3 can be also expressed with: *book1 rdf:type v:book; v:author "Fritz", "Egon"*.

We explain another notation for blank nodes with the following example: [*rdf:type v:book; v:author "Herbert", "Josef"*], where a predicate–object list occurs inside the brackets [] for blank nodes, is equivalent to [] *rdf:type v:book; v:author "Herbert", "Josef"*. The advantage of this notation is that additionally a predicate–object list on the right of the blank node brackets [] as well as a subject and predicate on the left of the blank node brackets [] are allowed. A subject and predicate on the left of the blank node brackets [] together with the new blank node as object form a triple; that is,

```
:bookshop1 v:sells
  [rdf:type v:book;
   v:author "Herbert", "Josef"]
   v:price "10"^^xsd:int
```

is equivalent to

```
:bookshop1 v:sells _:b.
_:b rdf:type v:book;
   v:author "Herbert", "Josef";
   v:price "10"^^xsd:int,
```

where *_:b* is a new blank node not used elsewhere.

A single-linked list is also called *collection* in N3. Collections in N3 (and also in SPARQL) are enclosed with brackets (and). For each element *e* of a collection, a new blank node *b* is generated. The blank node *b* and the current element *e* are connected via a triple (*b*, *rdf:first*, *e*). The blank node *b* is also connected with the blank node *b_{next}* for the next entry via a triple (*b*, *rdf:next*, *b_{next}*). For the last element in the collection, a triple (*b*, *rdf:next*, *rdf:nil*) is generated. Therefore,

```
("Friday" "Saturday" "Sunday")
```

is equivalent to

```
[rdf:first "Friday";
 rdf:next [rdf:first "Saturday";
           rdf:next [rdf:first "Sunday";
                   rdf:next rdf:nil ]]].
```

Again, analogous to the blank node brackets [], we can add a subject and a predicate on the left to the collection brackets () as well as a predicate–object list on the right of the collection brackets (), for example,

```
:list v:listOfDays ("Friday" "Saturday" "Sunday") v:altFirst "Freitag"@de.
```

is equivalent to

```
:list v:listOfDays [rdf:first "Friday";
                   rdf:next [rdf:first "Saturday";
                             rdf:next [rdf:first "Sunday";
                                       rdf:next rdf:nil ]]] v:altFirst "Freitag"@de.
```

The keyword *a* is an alias for *rdf:type*, the empty collection () for *rdf:nil*. The N3 notation contains more syntactic sugar. We refer the interested reader to (Berners-Lee 1998).

2.3.2 RDF/XML

RDF/XML uses XML to encode RDF triples. RDF/XML supports also a huge number of abbreviations and language constructs. We only present the RDF/XML representation (see Fig. 2.4) of the RDF data of Fig. 2.3 here and will not introduce the language constructs of RDF/XML further. We refer the interested reader to the specification of RDF/XML in (Beckett 2004) for more details. As N3 notation (or similar notations such as RDF triples or Turtle) is typically more human-readable than RDF/XML, one usually uses tools to transform RDF/XML files into one of these human-readable notations such as N3 before inspecting the content. These transformation tools are often part of an RDF store like Jena (McBride 2002; Wilkinson et al. 2003).

2.4 Ontology Languages

An ontology is meant to provide a standard description about a domain by defining a set of concepts, the properties of the concepts, and the relationships between these concepts. These relationships are either explicitly defined in the ontology, or can be

```

<rdf:RDF
  xmlns = "http://book/instances/"
  xmlns:v = "http://book/vocabulary/"
  xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about = "http://book/instances/book1">
    <rdf:type rdf:resource = "http://book/vocabulary/book"/>
    <v:author>Fritz</v:author>
    <v:author>Egon</v:author>
  </rdf:Description>
  <rdf:Description rdf:about = "http://book/instances/Buch2">
    <rdf:type rdf:resource = "http://book/vocabulary/Buch"/>
    <v:author>Fritz</v:author>
  </rdf:Description>
</rdf:RDF>

```

Fig. 2.4 RDF/XML representation of the RDF data of Fig. 2.3

asserted based on the existing ones from the ontology. This is the so-called inference capability of ontology. Ontology languages are formal languages used to construct ontologies. They allow the encoding of knowledge about specific domains and often include reasoning rules that support the processing of that knowledge. RDF Schema (RDFS) (Brickley and Guha 2004) and the Web Ontology Language (OWL) (Dean and Schreiber 2004) are two ontology languages developed by W3C.

RDFS provides basic elements for describing the ontologies in order to construct various RDF resources. RDFS allows the definition of classes, subclasses, and instances. It also allows the definition of properties, derived properties, and their domains and codomains. RDFS supports a type system and meta-classes.

OWL (Dean and Schreiber 2004) and its successor OWL 2 (Motik et al. 2009) allow more language constructs than RDF Schema. OWL (2) consists of three sublanguages OWL Lite, OWL DL, and OWL Full, with increasing expressiveness. OWL Lite contains language constructs for simple classification hierarchies and conditions, which allows simple and fast implementations. OWL DL is close to a syntactic variant of a more expressive, but still decidable (i.e., all computations are guaranteed to be completed in finite time), Description Logic (DL) (Baader et al. 2007), namely *SHOIN* (D). More precisely, the OWL DL variant coincides with this DL by imposing several restrictions on the usage of RDF(S) like disallowing meta-classes. Furthermore, OWL DL has the property of computation completeness; that is, the computation of all conclusions is guaranteed. The restrictions of OWL DL are lifted in OWL Full that combines the description logic flavor of OWL DL and the syntactic freedom of RDF(S). Figure 2.5 provides overviews of the

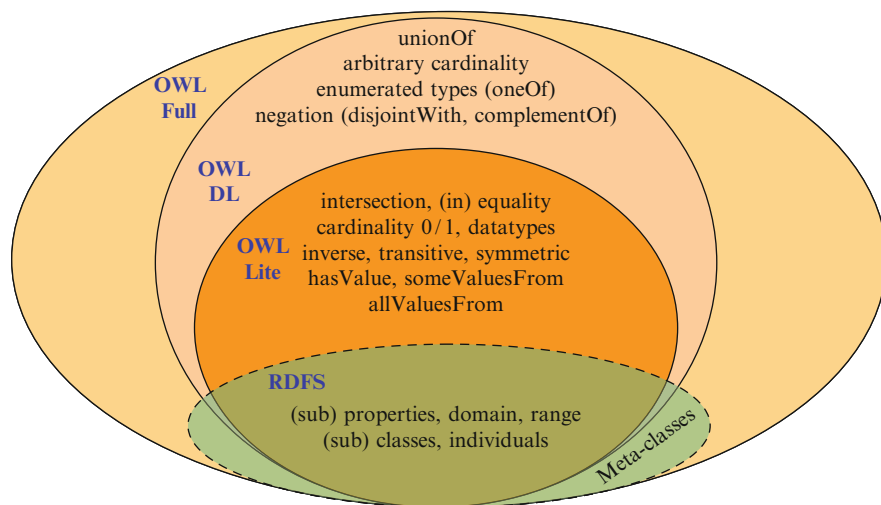


Fig. 2.5 Expressiveness of OWL sublanguages and RDFS

Fig. 2.6 Important OWL language constructs

Symmetric: if $P(x, y)$ then $P(y, x)$
Transitive: if $P(x, y)$ and $P(y, z)$ then $P(x, z)$
Functional: if $P(x, y)$ and $P(x, z)$ then $y = z$
InverseOf: if $P_1(x, y)$ then $P_2(y, x)$
InverseFunctional: if $P(y, x)$ and $P(z, x)$ then $y = z$
allValuesFrom: $P(x, y)$ and $y = \text{allValuesFrom}(C)$
someValuesFrom: $P(x, y)$ and $y = \text{someValuesFrom}(C)$
hasValue: $P(x, y)$ and $y = \text{hasValue}(v)$
cardinality: $\text{cardinality}(P) = N$
minCardinality: $\text{minCardinality}(P) = N$
maxCardinality: $\text{maxCardinality}(P) = N$
equivalentProperty: $P_1 = P_2$
intersectionOf: $C = \text{intersectionOf}(C_1, C_2, \dots)$
unionOf: $C = \text{unionOf}(C_1, C_2, \dots)$
complementOf: $C = \text{complementOf}(C_1)$
oneOf: $C = \text{one of}(v_1, v_2, \dots)$
equivalentClass: $C_1 = C_2$
disjointWith: $C_1 \neq C_2$
sameIndividuals: $I_1 = I_2$
differentFrom: $I_1 \neq I_2$
AllDifferent: $I_1 \neq I_2, I_1 \neq I_3, I_2 \neq I_3, \dots$
Thing: I_1, I_2, \dots

Legend:

Properties are indicated by: P, P_1, P_2 , etc
 Specific classes are indicated by: x, y, z
 Generic classes are indicated by: C, C_1, C_2
 Values are indicated by: v, v_1, v_2
 Instance documents are indicated by: I_1, I_2, I_3 , etc.
 A number is indicated by: N
 $P(x,y)$ is read as: "property P relates x to y "

expressiveness of RDFS and of OWL sublanguages, and Fig. 2.6 describes the important OWL language constructs.

OWL 2 additionally specifies 3 profiles: OWL 2 EL is particularly useful in applications employing ontologies that contain very large numbers of properties and/or classes. The basic reasoning problems for OWL 2 EL can be performed in time that is polynomial with respect to the size of the ontology. OWL 2 QL captures expressive power of simple ontologies such as thesauri, and (most of) expressive power of ER/UML schemas, which enables a tight integration with relational database management systems. OWL 2 RL is aimed at applications that require scalable reasoning without sacrificing too much expressive power. In the OWL 2 RL fragment, the ontology consistency, class expression satisfiability, class expression subsumption, instance checking, and conjunctive query answering problems can be solved in time that is polynomial with respect to the size of the ontology.

- (1) `@prefix v: <http://book/vocabulary/>.`
- (2) `@prefix owl: <http://www.w3.org/2002/07/owl#>.`
- (3) `@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.`
- (4) `owl:class rdf:ID v:book.`
- (5) `owl:class rdf:ID v:Book.`
- (6) `rdf:property rdf:ID v:author.`
- (7) `v:title owl:domain v:book.`
- (8) `v:book owl:equivalentClass v:Book.`

Fig. 2.7 An OWL ontology Book.owl in N3 notation

Multiple inheritances are allowed in RDFS as well as in OWL (2).

Figure 2.7 presents an OWL ontology Book.owl. The RDF data Book.rdf in Fig. 2.3 is an instance data of this ontology. Book.owl defines three concepts, *v:book*, *v:Book*, and *v:author*, and several relationships: *v:book* and *v:Book* are classes [lines (4) and (5)]; *v:author* is a property [line (6)] of *v:book* [line (7)]; and *v:book* and *v:Book* are equivalent classes [line (8)]. A number of implied relationships can be asserted; for example, if a book is an instance of *v:book*, then it is also an instance of *v:Book*.

2.5 Open World Assumption

In this and in the next subsection, we want to introduce some paradigms, which are realized in the Semantic Web technologies and which differ from known paradigms, for example, from the database world. These new paradigms address new challenges in web environments: data is often incomplete, as new data sources occur frequently, data sources are temporarily unavailable, or disappear permanently.

The first paradigm is the *Open World Assumption*, which is contrary to the well-known *Closed World Assumption*. Recall that the Closed World Assumption states that a database contains all facts and that everything not contained in the database is assumed to be false. Contrary, the Open World Assumption assumes everything not contained in the database *not* to be false, but to be *unknown*! If a data source contains, for example, only the information that trains depart at 2 and 5 p.m., then a query about the existence of a train departing at 4 p.m. would be answered using the Closed World Assumption with *false*. However, using the Open World Assumption, the answer should be *unknown*, as there could be another data source containing this information. OWL reasoning follows the Open World Assumption and as consequence, a fact that a train departs at 4 p.m. is *not* inconsistent with the previous given information that trains depart at 2 and 5 p.m.

2.6 No Unique Name Assumption

In the database world, every resource (like a table row in a relational database) has an identifier (consisting of the attribute values of the primary key in a relational table). This resource has neither another identifier nor has another resource the same identifier. In an open context like the Web, this cannot be guaranteed and the uniqueness of identifiers must be lifted. In the No Unique Name Assumption several different identifiers can refer to the same resource. This affects reasoning, too. For example, the facts that a person Mary has the parents Lisa Minelly, John Doe, and Lisa Doe are given. Furthermore, we have a cardinality axiom stating that a person has at most two persons as parents. In the Unique Name Assumption as used in the database world, it would be a contradiction to have the parents Lisa Minelly, John Doe, and Lisa Doe, which are more than two. In the No Unique Name Assumption as used in OWL reasoning, it can be inferred that Lisa Minelly is the same person as John Doe or as Lisa Doe, or John Doe is the same person as Lisa Doe, or Lisa Minelly, John Doe and Lisa Doe are all one person. For a human being, it is intuitive that Lisa Minelly and Lisa Doe is the same person, as Lisa Minelly could be the maiden name of Lisa Doe. A reasoner can only determine this fact if much more information is given such as Lisa Doe and Lisa Minelly are females, John Doe is male, and female is disjoint from male.

2.7 SPARQL Query Language

Just as SQL is the most important query language for relational databases, SPARQL (Prud'hommeaux and Seaborne 2008; Beckett and Broekstra 2008; Clark et al. 2008) is the most important query language for the Semantic Web.

Before SPARQL has been recommended as standard RDF query language by the W3C, many proprietary RDF query languages have been developed. Among them are RDQL (Seaborne 2004) and N3QL (Berners-Lee 2004). RDQL was influenced by SquishQL and rdfDB (Miller et al. 2002), while N3QL was influenced by Notation 3 (Berners-Lee 1998) and RDQL. Furthermore, TAP (Guha and McCool 2003) has been specified for the semantic search, and Sesame RDF Query Language (SeRQL) (Broekstra and Kampman 2003) has been introduced as part of Sesame (Broekstra et al. 2002). YARSQL (Harth and Decker 2005) is a quadruple query language, which additionally considers the context of a triple (see Guha et al. 2004; MacGregor and Ko 2003). Today, SPARQL is accepted as standard query language and is supported by main database vendors such as Oracle (Oracle 2009).

SPARQL offers a powerful means to query RDF triples and graphs and supports a variety of querying capabilities. Results of SPARQL queries can be ordered, limited and offset by a given number of elements. The W3C has plans for embedding SPARQL into other W3C languages, analogous to the embedding of the XML query language XPath (W3C 2007b) into XQuery (W3C 2007c) and XSLT (W3C 2007a).

The core component of SPARQL queries is a set of triple patterns $s p o . s p o$ corresponds to the subject (s), predicate (p), and object (o) of an RDF triple, but they can be variables as well as RDF terms. Within a SPARQL query, the user specifies the known RDF terms of triples and leaves the unknown ones as variables in triple patterns. The same variables can occur in multiple triple patterns and thus imply joins. A triple pattern matches a subset of the RDF data, where the RDF terms in the triple pattern correspond to the ones in the RDF data. A triple pattern applied to an RDF graph generates an unordered bag of solutions. A solution is a set of bindings, each of which consists of a pair of a variable and its bound value, that is, corresponding RDF terms in the matched subset of the RDF data. The result of a group of triple patterns is the join of the results of individual triple patterns.

There are several limitations on SPARQL concerning the supported language constructs. For example, SPARQL allows the nesting of subexpressions only up to a certain degree (e.g., a SPARQL construct query, the result of which is an RDF graph, cannot be the input of an outer SPARQL query) and does not support path expressions (e.g., to retrieve the descendants of a node), the expressions to compute the transitive closure, updates, user-defined functions, or rules.

Proprietary extensions of SPARQL exist such as SPARUL (Seaborne and Manjunath 2008) for supporting update operations. SPARUL and some other extensions like GROUP BY, aggregation functions, and nested subqueries are currently in the standardization process for SPARQL 1.1 (Kjernsmo and Passant 2009).

2.7.1 Language Constructs of SPARQL

Before describing the main aspects of the SPARQL query language, we first present an example SPARQL query in order to provide readers a first flavor of SPARQL queries.

Figure 2.8 is an example of a SPARQL query `Book.sparql`, which can be applied on the RDF data `Book.rdf` in Fig. 2.3. `Book.sparql` contains a **SELECT** clause in line (3) and a **WHERE** clause in lines (4) and (5). The **SELECT** clause identifies the variables to appear in the query results (i.e., the bindings of the variable `?author`). The **WHERE** clause contains two triple patterns, which identify the constraints on the input RDF data. The first triple pattern `?x v:author ?author` matches three triples of `Book.rdf` (see lines (5), (6), and (8) in Fig. 2.3), and thus its result is $\{(?x=:book1, ?author="Fritz"), (?x=:book1, ?author="Egon"), (?x=:Buch2, ?author="Fritz")\}$. Without considering the ontology `Book.owl` in Fig. 2.7, the

```
(1) PREFIX v: <http://book/vocabulary/>
(2) PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
(3) SELECT ?author
(4) WHERE { ?x v:author ?author .
(5)           ?x rdf:type v:book. }
```

Fig. 2.8 A SPARQL query `Book.sparql`

second triple pattern `?x rdf:type v:book` only matches one triple of `Book.rdf`, that is, `(:book1, rdf:type, v:book)` (see line (4) in Fig. 2.3), and thus its result is `{(?x=:book1)}`. The two triple patterns impose a join over the common variable `?x`, and the join result of the two triple patterns is hence `{(?x=:book1, ?author="Fritz"), (?x=:book1, ?author="Egon")}`. The final query result is a bag `{(?author="Fritz"), (?author="Egon")}`. The statement, `v:book owl:equivalentClass v:Buch`, in the ontology `Book.owl` in Fig. 2.7 [see line (8)] implies that an instance of `v:Buch` is also an instance of `v:book`. Therefore, we can infer that `:Buch2` is an instance of `v:book`; that is, the triple `(:Buch2, rdf:type, v:book)` can be added to `Book.rdf` as implicit knowledge. When considering this ontology information, the result of the second triple pattern is hence `{(?x=:book1), (?x=:Buch2)}`, and thus the final query result is `{(?author="Fritz"), (?author="Egon"), (?author="Fritz")}`.

2.7.1.1 Types of SPARQL Queries

There are four types of SPARQL queries.

The most often used type is the **SELECT** query, which we have just introduced above. The bindings of all the variables occurring in a query are returned when using the wildcard `*` instead of a projection list of variables. A projection list is a parameter of the **SELECT** clause and consists of variable names [see line (3) in Fig. 2.8]. By replacing line (3) of Fig. 2.8 with **SELECT** `?x ?author` or with **SELECT** `*`, we retrieve `{(?x=:book1, ?author="Fritz"), (?x=:book1, ?author="Egon")}` when applying the query on the RDF data in Fig. 2.3.

Additionally to the **WHERE** clause like in **SELECT** queries, **CONSTRUCT** queries have a set of triple templates. For each solution of the **WHERE** clause, variables in the triple templates are substituted with the bound values in the solution, such that RDF triples are generated. All generated RDF triples build an RDF graph, which is the result of the **CONSTRUCT** query. For example, the result of the **CONSTRUCT** query in Fig. 2.9 is the set of triples `{(:book1, v:author, "Fritz"), (:book1, v:author, "Egon"), (:book1, rdf:type, v:Buch)}` when applied on the RDF data in Fig. 2.3.

Users can apply **ASK** queries in order to test whether or not a given query pattern has a solution. An **ASK** query returns a Boolean value indicating whether or not a solution exists. For example, replacing line (3) in Fig. 2.8 with **ASK** and applying this query on the RDF data in Fig. 2.3 would return `true`.

```

PREFIX v: <http://book/vocabulary/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
CONSTRUCT {
    ?x v:author ?author .
    ?x rdf:type v:Buch .
WHERE {
    ?x v:author ?author .
    ?x rdf:type v:book .
  }

```

Fig. 2.9 Construct query

DESCRIBE queries return RDF graphs containing relevant information about the resources found for a given query pattern in the **WHERE** clause. However, the result of a DESCRIBE query has not been formally defined and is implementation-dependent. In most implementations, replacing line (3) in Fig. 2.8 with **DESCRIBE** *?x* and applying this query on the RDF data in Fig. 2.3 would return at least all triples containing *:book1*, that is, $\{(:book1, v:author, "Fritz"), (:book1, v:author, "Egon"), (:book1, rdf:type, v:book)\}$.

2.7.1.2 Default Graph and Named Graphs

An RDF store applies a query on a certain RDF graph, the *default RDF graph*, if no other RDF graphs are addressed in the query. Besides a default RDF graph, an RDF store administers different RDF graphs and uses IRIs to distinguish them. SPARQL uses the **FROM IRI** clause to indicate a nondefault RDF graph with name *IRI* to be queried. For example, we assume that the triples in lines (4)–(6) of Fig. 2.3 are stored in an RDF graph associated with the IRI $\langle http://www.graph1.com \rangle$ in the RDF store, and the triples in lines (7) and (8) of Fig. 2.3 are stored in another RDF graph associated with the IRI $\langle http://www.graph2.com \rangle$. We also use these RDF graphs in other examples of this subsection. When adding

FROM $\langle http://www.graph1.com \rangle$

FROM $\langle http://www.graph2.com \rangle$

between the lines (3) and (4) of Fig. 2.8, the query is evaluated on the union of both RDF graphs.

SPARQL even supports to evaluate different triple patterns on different RDF graphs within the same query. For this purpose, the **FROM NAMED IRI** clause specifies the RDF graph with name *IRI* to be used as *named graph* rather than as default RDF graph like for a **FROM IRI** clause. Then a **GRAPH IRI { E }** clause indicates that the inner query expression *E* in the curly brackets will be evaluated on the named RDF graph *IRI*. If a variable, for example, *?v* rather than an *IRI* is used, the inner query expression *E* is evaluated on all named graphs. The variable *?v* will be bound with the IRI of the corresponding named RDF graph. For example, the query in Fig. 2.10 has the result $\{(?author="Fritz", ?x=:book1, ?t=v:book,$

```

PREFIX v: <http://book/vocabulary/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT *
FROM NAMED <http://www.graph1.com>
FROM NAMED <http://www.graph2.com>
WHERE { GRAPH ?g {
    ?x v:author ?author.
    ?x rdf:type ?t. }}

```

Fig. 2.10 SPARQL query using named graphs

?g=<http://www.graph1.com>), (*?author="Egon"*, *?x=:book1*, *?t=v:book*,
?g=<http://www.graph1.com>), (*?author="Fritz"*, *?x=:Buch2*, *?t=v:Buch*,
?g=<http://www.graph2.com>)}

2.7.1.3 Other Modifiers

All query types except of the ASK queries support the modifier **ORDER BY** for sorting the result of queries, by a given number limited query result when using the **LIMIT** modifier, and considering an offset in the query result when using the **OFFSET** modifier.

The **ORDER BY** clause causes sorted query results. After the **ORDER BY** keyword, a list of sort criteria can be given. The first sort criterion in this list is the primary sort criterion, the second in this list the secondary sort criterion, and so on. Each sort criterion can start with **ASC** for ascending order or **DESC** for descending order. If neither **ASC** nor **DESC** is given, then the ascending order is assumed. A variable may be given as sort criterion. It is also possible to specify a more complex expression as sort criterion like a built-in function or a formula like $?x + ?y$, where the solutions of the query are sorted according to the result of the given expression applied on the solutions.

The **OFFSET** x clause discards the first x solutions from the query result, where x is an integer.

Complementary to the **OFFSET** clause, when using the **LIMIT** x clause, only the first x solutions remain in the query result and the others are discarded, where x is an integer.

LIMIT and **OFFSET** clauses can be combined, that is, when using **OFFSET** x **LIMIT** y , the $(x+1)$ th to $(x+y+1)$ th query result solutions remain and the others are discarded.

For example, when adding the following lines

ORDER BY *?author*

LIMIT 1

OFFSET 1

to the query in Fig. 2.8 after line (5), we retrieve the result $\{(?author="Fritz")\}$ when evaluating the modified query on the RDF data in Fig. 2.3.

2.7.1.4 Variables and Blank Nodes

Variables are placeholders for the RDF terms in RDF data. Solutions of a query result contain bindings of variables with these RDF terms. Variables start with a dollar character \$ or a question mark ?, followed by the name n of the variable, for example, $\$n$ and $?n$. Variables with the same name n represent the same variable; that is, $\$n$ and $?n$ are the same variables!

Sometimes the user does not want to include a variable in the query result. In such a case, the user can use a SPARQL blank node: The SPARQL language

redefines the semantics of blank nodes of RDF using the same notation as in N3, that is, [] for a new blank node without name and *_:b* for a blank node with name *b*. Blank nodes in SPARQL are local variables, whereas blank nodes in RDF are RDF terms with (local) identifiers. Blank nodes in SPARQL can neither be used in the projection list of SELECT queries nor their bound values appear in the query result even if the wildcard *** is used in a SELECT clause. For example, the query in Fig. 2.11 returns the same result as the query in Fig. 2.8.

2.7.1.5 Triple Patterns

For triple patterns, SPARQL supports abbreviations for object lists, predicate–object lists, blank nodes, and collections. These abbreviations are similar to the ones for the N3 notation and only differ in the additional support of variables besides RDF terms. For example, the triple patterns in Fig. 2.11 may be replaced with `[v:author ?author; rdf:type v:book]`, which does not change the query result for any input RDF graph.

As in the N3 notation, the keyword *a* represents *rdf:type* and the empty collection () stands for *rdf:nil*.

2.7.1.6 Filter

Filter expressions contain Boolean formulas. For each solution of the intermediate query result, a filter expression checks whether its Boolean formula becomes true, false, or an error occurs like a type error or that a used variable is not bound. If its Boolean formula becomes true, then the solution remains in the intermediate query result, otherwise in the case that the Boolean formula becomes false or an error occurs, the solution is discarded. A Boolean formula is an RDF term or a variable. Furthermore, if *B1* and *B2* are Boolean formulas, then the negated formula *!B1*, formulas containing logical and- (*B1 && B2*) and or- (*B1 || B2*) combinations, relational operations (*B1 < B2*, *B1 <= B2*, *B1 = B2*, *B1 > B2*, *B1 >= B2*, *B1 != B2*), mathematical formulas such as *B1 + B2*, or built-in functions are also Boolean formulas. A Boolean formula transforms other values than the Boolean values *true* or *false* and errors by determining the effective Boolean value (see Prud'hommeaux and

```

PREFIX v: <http://book/vocabulary/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT *
WHERE { _:b v:author ?author.
          _:b rdf:type v:book. }

```

Fig. 2.11 Query using blank nodes

Seaborne 2008). For example, the effective Boolean formula of a plain literal or a typed literal with a data type of *xsd:string* is false if the result has zero length; otherwise the effective Boolean value is true. The effective Boolean value of a numeric type or a typed literal with a data type derived from a numeric type is false if the operand value is *NaN* (alias for *not a number*) or is numerically equal to zero; otherwise the effective Boolean value is true.

For example, by inserting ***FILTER***(*?author*="Fritz") between lines (4) and (5) in Fig. 2.8, we would retrieve the query result `{(?author="Fritz")}`.

2.7.1.7 Built-In Functions

SPARQL supports various built-in functions to be used in Filter expressions. We focus on the important ones here, whereas a complete list of built-in functions is given in (Prud'hommeaux and Seaborne 2008).

The function *bound* checks whether or not a variable is bound. The functions *isIRI* (equivalent to the function *isURI*), *isBlank*, and *isLiteral* check whether or not a RDF term is an IRI, blank node, or a literal, respectively. The function *datatype* returns the data type of a literal. In the case of a simple literal, *xsd:string* is returned as data type. The function *REGEX* checks whether or not a regular expression is matched. *REGEX* is actually equivalent to the XPath *fn:matches* function as defined in Malhotra et al. (2007). SPARQL further imports a subset of the XPath constructor functions defined in (Malhotra et al. 2007). A constructor function as, for example, *xsd:long* allows to cast a value to the corresponding data type of the constructor function. Data type-specific operations such as comparisons or mathematical operations can be afterward calculated using the casted values, even though the values themselves are simple literals or have another data type.

For example, by inserting ***FILTER***(*REGEX*(*?author*, "E")) between lines (4) and (5) in Fig. 2.8 for checking if *?author* contains *E*, we would retrieve the query result `{(?author="Egon")}`.

RDF stores typically support more functions than only the built-in functions of the specification and also support to call external functions programmed in another programming language like Java [as Jena (McBride 2002) (Wilkinson et al. 2003) does].

2.7.1.8 Optional

The *OPTIONAL* operator adds bindings of its right operand to the solutions of its left operand if the right operand matches and do not bind other values to the same variables as in the solution of the left operand; otherwise the *OPTIONAL* operator let the solution of the left operand unchanged; that is, the effect of an *OPTIONAL* operator is like for a left-outer join in relational databases.

Furthermore, the *OPTIONAL* operator in combination with a ***FILTER***(*!bound*(*?v*)) can simulate *Negation as Failure* in logic programming and a *NOT EXISTS*

```

PREFIX v: <http://book/vocabulary/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x
WHERE { ?x v:author ?author.
          OPTIONAL { ?x rdf:type ?t.
                     FILTER(?t=v:book).}
          FILTER(!bound(?t)) }

```

Fig. 2.12 Example of negation as failure in SPARQL

```

PREFIX v: <http://book/vocabulary/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT *
WHERE { ?x v:author ?author.
          {?x rdf:type v:book .} UNION {?x rdf:type v:Buch .} }

```

Fig. 2.13 SPARQL query with *UNION* operator

clause of the upcoming SPARQL 1.1 standard. For example, the query in Fig. 2.12 returns all those RDF terms that have an author and are not of type *v:book*. When evaluating the query in Fig. 2.12 on the RDF data in Fig. 2.3, we get the result $\{(?x=:Buch2)\}$. In more detail, the pattern *A OPTIONAL {B} FILTER(!bound(?v))*, where *A* and *B* are SPARQL subexpressions, and the variable *?v* must occur in *B* for every result of *B* and not in *A*, can be used in SPARQL 1.0 to retrieve all solutions of *A* for which *B* does not have any solutions.

2.7.1.9 Union

The *UNION* operator returns all the solutions of all its operands. For example, the query in Fig. 2.13 uses the *UNION* operator to retrieve all those RDF terms of type *v:book* or *v:Buch*, which have an author. The result of this query applied on the RDF data in Fig. 2.3 is hence $\{(?author="Fritz", ?x=:book1), (?author="Egon", ?x=:book1), (?author="Fritz", ?x=:Buch2)\}$.

2.7.2 SPARQL Protocol for RDF

To increase the interoperability of different SPARQL engines, (Clark 2008) proposes a protocol, which specifies how to set up queries at a SPARQL engine over the internet. SPARQL engines supporting the protocol specified in (Clark 2008) are also called *SPARQL endpoints*. The SPARQL protocol uses WSDL 2.0 (Chinnici et al. 2007) to describe how to convey SPARQL queries to a SPARQL endpoint and return the query results to the requesting entity. The SPARQL Protocol is described

in an abstract interface independent of any concrete realization, implementation, or binding to another protocol. Furthermore, HTTP and SOAP bindings of this interface are provided. We do not explain all the details here and just provide a simple example: Fig. 2.14 contains the SOAP message for setting up the query of Fig. 2.8. Another SOAP message (see Fig. 2.15) is used to transport the query result

```

POST /services/sparql-query HTTP/1.1
<?xml version = "1.0" encoding = "UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv = "http://www.w3.org/2003/05/soap-envelope/"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
<soapenv: Body>
<query-request
  xmlns = "http://www.w3.org/2005/09/sparql-protocol-types/#">
  <query>
    PREFIX v: <http://book/vocabulary/>
    PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    SELECT ?author
    WHERE{
      ?x v:author ?author.
      ?x rdf:type v:book.}
  </query>
</query-request>
</soapenv:Body>
</soapenv:Envelope>

```

Fig. 2.14 SOAP message for setting up the query of Fig. 2.8

```

<?xml version = "1.0" encoding = "utf-8"?>
<soapenv:Envelope
  xmlns:soapenv = "http://www.w3.org/2003/05/soap-envelope/"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
  <query-result xmlns = "http://www.w3.org/2005/09/sparql-protocol-types/#">

    RESULT

  </query-result>
</soapenv:Body>
</soapenv:Envelope>

```

Fig. 2.15 SOAP message containing the query result

back. *RESULT* in Fig. 2.15 is a placeholder for the query result. The query result itself follows a special format, which we explain in the next subsection.

2.7.3 SPARQL Query Results XML Format

For the interoperability of different SPARQL engines, we have to not only specify how to set up queries at SPARQL engines over the internet, but also specify the format of the query result. For this purpose, (Beckett and Broekstra 2008) specify an XML format for the variable binding and Boolean results of SELECT queries and ASK queries called *SPARQL Query Results XML Format*. Note that CONSTRUCT and DESCRIBE queries return an RDF graph, which can be serialized in XML using RDF/XML (Beckett 2004).

For example, the query result $\{(?x=:book1, ?author="Fritz"), (?x=:book1, ?author="Egon")\}$ can be expressed as in Fig. 2.16. The `<head>` tag contains all variables in the query result; the `<results>` tag contains all solutions of the query. Each solution is contained in one `<result>` tag. A variable binding of a variable n is contained in a `<binding name="n">` tag. The bound values can be encoded in an `<uri>` tag in the case of an URI, in a `<literal>` tag in the case of a literal, or in a `<bnode>` tag in the case of a blank node. The `<literal>` tag can have no attribute for simple literals, the attribute `datatype="D"` for a typed literal with data type D , or the attribute `xml:lang="L"` for a language-tagged literal with language L .

```
<?xml version = "1.0"?>
<sparql xmlns = "http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name = "x"/>
    <variable name = "author"/>
  </head>

  <results>
    <result>
      <binding name = "x"> <uri>http://book/instances/book1</uri></binding>
      <binding name = "author"> <literal>Fritz</literal></binding>
    </result>

    <result>
      <binding name = "x"> <uri>http://book/instances/book1</uri></binding>
      <binding name = "author"> <literal>Egon</literal></binding>
    </result>
  </results>
</sparql>
```

Fig. 2.16 SPARQL query results format of $\{(?x=:book1, ?author="Fritz"), (?x=:book1, ?author="Egon")\}$

2.7.4 RDF Stores

We classify storage systems according to the logical and physical data model and the supported query languages. Structured data are typically stored with support of indices for fast access to the data. RDF storage systems provide functionalities like support of permanent RDF data, updates, and deletion. Figure 2.17 contains an overview of several RDF storage systems.

RDF storage system	Data model	Storage model	Supported query languages	Data access interface	Inference of new facts
Jena (McBride 2002; Wilkinson et al. 2003)	Triple	RDBMS, File	SPARQL, RDQL, Triple Pattern	RDF/XML, N-Triples, N3, Turtle, NetAPI	Rule based
YARS (Harth and Decker 2005)	Quadruple	B+ tree	Tripel Pattern, YARSQL	N-Triples, HTTP	-
SESAME (Broekstra et al. 2002)	Triple	Abstract, RDBMS, File	SPARQL, RQL, RDQL, SeRQL	RDF input/output library, RDF/XML, N-Triple, N3, Turtle	Rule based
Oracle Spatial 11g (Oracle, 2009)	Triple	Oracle Spatial Network Data Model	Extension of standard SQL, planned: SPARQL	Extension of standard SQL	Inference based on RDFS, big subset of OWL
ORDI (Kiryakov et al. 2004)	Triple	SESAME	SPARQL, RQL, RDQL, SeRQL	RDF input/output library, RDF/XML, N-Triple, N3, Turtle	Rule based
3Store (Harris and Gibbins 2003)	Triple	RDBMS	SPARQL, RDQL	RDF/XML, N-Triples, Turtle, HTTP	Inference based on RDFS
Kowari (Wood et al. 2005)	Quadruple	AVL Tree	iTQL, planned: SPARQL	RDF/XML, N3, SOFA, JRDF, SOAP	DL, OWL-Lite
RDFPeers (Sung et al. 2005)	Triple	RDBMS	RDQL, RDFPeers queries	RDF/XML	-
Edutella (Edutella 2004)	Triple	RDBMS	RDF-QEL (adaptet from Datalog)	RDF/XML	Adapted from Datalog

Fig. 2.17 Overview of existing RDF storage systems

2.8 Rules

Rules have been used, for example, to control devices and processes in real-time applications, perform calculations or inference, enforce integrity constraints on databases, represent and enforce policies, and determine the need for human intervention. The RIF rule language has been developed by the W3C in order to support advanced reasoning capabilities by integrating it with ontology languages.

Ontology languages describe knowledge according to the Open World Assumption: the encoded knowledge is considered incomplete, and the conclusions, which cannot be derived from ontologies, are treated agnostically. Rule inference as proposed by RIF follows the Closed World Assumption, where everything which is not derivable is assumed to be false. This allows reasoning, for example, in domains that have to deal with default knowledge, that is, knowledge that usually holds like “birds typically fly”, unless there is evidence of the contrary. Therefore, rule languages are introduced as complementation for ontologies.

Today’s state-of-the-art applications run in distributed and heterogeneous environments (communicating, e.g., using the internet). Using rules in this situation requires a widely accepted and supported standard for rules interchange, such that rules can be processed by different distributed systems running on different platforms and using different rule engines.

However, before RIF, there was neither a common standard for rules interchange nor a rule language specialized for the Semantic Web. The language for RIF rules is standardized by the W3C, the world’s leading standardization committee for the Web. RIF rules – in comparison to prolog and datalog rules, are specialized for the usage in the Semantic Web. This opens new possibilities and additional advantages for Semantic Web applications, for example, more interchangeability, more concisely processing by additionally considering the semantics based on ontologies, and a widespread support of further Semantic Web technologies.

Logic-based dialects cover languages that apply some kind of logic, such as first-order logic (often restricted to Horn logic) or non-first-order logics, which underlay the various logic programming languages like logic programming using the well-founded (Gelder et al. 1991) or stable (Gelfond and Lifschitz 1988) semantics. The rules-with-actions dialects include production rule systems, such as Drools,¹ Jess,² and JRules,³ as well as reactive (or event-condition-action) rules, such as Reaction RuleML⁴ and XChange.⁵

The RIF working group aims to provide representational interchange formats for processes based on the use of rules and rule-based systems. These formats act as

¹<http://jboss.org/drools/>

²<http://www.jessrules.com/>

³<http://www.ilog.com/products/jrules/>

⁴<http://reaction.ruleml.org/>

⁵<http://reactiveweb.org/xchange/>

RIF BLD	RIF CORE	RIF PRD
<ul style="list-style-type: none"> • Equality in conclusions • membership in conclusions • External Functions • Frame subclass • Open Lists • “logic” functions 	<ul style="list-style-type: none"> • Horn (monotonic) • Datatypes & builtins • external functions • frames, class membership (in conditions) • equality (in conditions) • ground lists • existential quantification (in conditions) 	<ul style="list-style-type: none"> • Conclusion “actions” • Negation • frames-as-objects • Retraction • subclass • membership in conclusion

Fig. 2.18 Expressiveness of RIF dialects, adapted from http://www.w3.org/2005/rules/wiki/images/b/b0/W3C_RIF-CW-9-09.pdf

- (1) **Document**(*Prefix*(cpt <http://example.com/concepts#>)
- (2) *Prefix*(ppl <http://example.com/people#>)
- (3) *Prefix*(bks <http://example.com/books#>)
- (4) **Group**(
- (5) *Forall* ?Buyer ?Item ?Seller(
- (6) *cpt:buy*(?Buyer ?Item ?Seller):-
- (7) *cpt:sell*(?Seller ?Item ?Buyer))
- (8) *cpt:sell*(ppl:John bks:LeRif ppl:Mary))

Fig. 2.19 Simple complete RIF Core example [taken from (Boley et al. 2009)]

“interlingua” to exchange rules and integrate with other languages, in particular (Semantic) Web markup languages.

The RIF languages are designed for two main kinds of dialects (see Fig. 2.18): *logic-based dialects* [e.g., the RIF Core Dialect (Boley et al. 2009) and the Basic Logic Dialect (RIF-BLD) (Boley and Kifer 2009)], and dialects for *rules with actions* [e.g., the Production Rule Dialect (RIF-PRD) (Sainte Marie et al. 2009)]. Both dialects RIF-BLD and RIF-PRD extend the RIF Core Dialect (Boley et al. 2009). Other dialects are expected to be defined by the various user communities. Figure 2.18 provides an overview of the expressiveness of the different RIF dialects. RIF Core is basically a syntactic variant of Horn rules, which most available rule systems can process. RIF allows frames as in F-Logic notation, is compatible with the RDF data model, supports the use of IRIs (Dürst and Suignard 2005) as object identifiers (where IRIs are enclosed in angle brackets), and typed literals. For instance, the RDF triple (<http://book/instances/book1>, dcterms:issued, "2006"^^xsd:gYear) can be represented as a RIF frame <http://book/instances/book1> [dcterms:issued -> "2006"^^xsd:gYear]. RIF uses the Prolog style “:-” for separating rule head (called *left side* or *consequent*) and body (called *right side* or *antecedent*).

The example in Fig. 2.19 presents a simple complete RIF Core rule [taken from (Boley et al. 2009)], which describes several buy–sell relationships in order to

```

Document( Prefix( v <http://book/vocabulary/> )
           Prefix( rdf <http://www.w3.org/1999/02/22-rdf-syntax-ns#> )
Group(
  Forall ?author(
    v:bookauthors(?author) :- AND(
      ?x[v:author->?author]
      ?x[rdf:type->v:book] ) ) ) )

```

Fig. 2.20 Rule for retrieving the book titles analogous to the SPARQL query in Fig. 2.8

derive new buy–sell relationships (1) A buyer buys an item from a seller if the seller sells the item to the buyer [see lines (5)–(7)]. (2) John sells LeRif to Mary [see line (8)]. The fact *Mary buys LeRif from John* can be logically derived by a *modus ponens* argument, which can be determined by application of the rule in lines (5)–(7). *Groups* [see line (4)] allow to associate rules with a priority and a resolution conflict strategy. The current RIF Core specification (Boley et al. 2009) defines only the resolution conflict strategy *forward-chaining*, which is the default. Recall that forward-chaining starts with the available data and applies given rules to infer new data until a goal is reached. The lines (1)–(3) contain prefix declarations similar to SPARQL queries. Figure 2.20 contains another RIF example for retrieving the book’s authors analogous to the SPARQL query in Fig. 2.8. In this example, the frames `?x[v:author->?author]` and `?x[rdf:type->v:book]` are used analogously to the triple patterns of the SPARQL query in Fig. 2.8.

RIF rules specify how the RIF rules themselves interoperate with RDF graphs and RDFS/OWL ontologies, and specify the conditions under which the combination of RIF rules, RDF graphs, and ontologies is satisfiable (i.e., consistent), as well as the entailments (i.e., logical consequences based on inference). The interaction between RIF and RDF/OWL is realized by connecting the model theory of RIF with the model theories of RDF (Hayes 2004) and OWL (Motik et al. 2009). When RDF graphs are imported into RIF, it must be specified whether the satisfiability or entailment of a model (Simple, RDF, D (for data type support), OWL DL, or OWL Full) is the basis for the combination of RIF with the imported data. Bruijn (2009) provides more information about these combinations.

RIF supports the XML Schema data types (Peterson et al. 2009) and various functions: a huge amount of built-in functions for comparing values, data type functions for checking if a literal is (or is not) of a certain data type, data type conversions and castings functions, (basic) numeric functions, Boolean functions, and functions on strings, dates, times, durations, xml literals, plain literals, and lists.

Rules in RIF-PRD support actions as consequent in the (production) rules. The actions can be assertions and retractions of facts and frames, modifications (i.e., additions, removals, or replacements) of frames, executions of externally defined actions, or sequences of these actions, including the declaration of local variables and a mechanism to bind a local variable with a frame slot value or a new frame object.

A typical scenario for the use of RIF with RDF/OWL is the exchange of rules that use RDF data and/or RDFS or OWL ontologies. An interchange partner *A* uses a rule language that is RDF/OWL-aware; that is, the input data of the rules are RDF data, an RDFS or OWL ontology defines the semantics of objects used in the rules, or the rules extend RDF(S)/OWL inference. *A* transmits its rules using RIF, possibly with references to input RDF graph(s), to partner *B*. *B* receives the rules and retrieves the referenced RDF graph(s). The rules are then processed together with the retrieved RDF graphs.

A specialization of the previous scenario is the publication of RIF rules: a publisher publishes its rules in the Web and consumers retrieve the RIF rules (and referenced RDF graphs) from the Web and process the retrieved RIF rules together with the RDF graphs in their own rule engines.

In another exchange scenario, the intention of a publisher is to extend an OWL ontology with rules. This publisher splits its ontology and rules description into a separate OWL ontology and a RIF document that includes a reference to the OWL ontology and publishes them. A consumer of the rules retrieves the OWL ontology and RIF document and translates both into a description, which combines ontology and rules, in its own rule extension of OWL.

There are a number of systems available or planned for RIF dialects (see Fig. 2.21).

2.9 Related Work

2.9.1 RIF Processing

As RIF only recently appeared, there are only few publications concerning RIF processing. A RIF tutorial is given in Marie (2008), and Hawke (2009) and Kiefer (2008) are two keynotes about RIF.

There are several demonstrations (Bost et al. 2007; Hallmark et al. 2008; IBM 2008), which describe systems supporting RIF. In (Bost et al. 2007), the Mortgage Industry Maintenance Organization (MISMO) describes a proof-of-concept (POC) to solve an often mentioned need for the exchange of rules: loan application pricing. Bost et al. (2007) develop an extended Production Rules (PR) language of the Rules Interchange Format (RIF). In Bost et al. (2007), the well-established MISMO schema is used as the ontology for sharing and executing a rule set among ILOG JRules and JBOSS Rules in a distributed environment. Hallmark et al. (2008) describe a system, where rules are interchanged between the rule engines such as ILOG JRules, Oracle, and Prova. IBM (2008) contains a web demo for RIF processing using the IBM rule engine.

(continued)

Rule System	Organization, Contact	RIF dialect	Consumer, Producer, Function	Impl. language & license	Time frame
SILK	Vulcan, BBN, Stony Brook University, B. Grosz, M. Dean, M. Kifer	BLD + development of a Default Logic Dialect (DLD) extending BLD	producer, consumer	Java (license TBD)	October (BLD) and December (DLD) 2009
OntoBroker 5.3	ontoprise Christian Schmidt	BLD	producer, consumer	commercial	Available
fuxi	Chimezie Ogbuji	RIF Core/OWL 2 RL	producer	BSD license	Available
N/A	Susan Malaika	BLD (but should support all XML syntaxes)	producer	IBM DeveloperWorks	Available
IBM Websphere ILOG JRules	Changhai Ke IBM/ILOG	PRD+DTB	producer-consumer		October 2009
Eye	Jos De Roo	BLD+DTB using external RIF/XML->N3 translator	consumer	Yap and W3C License	DTB available
Vampire Prime	Alexandre Riazanov	BLD	consumer	LGPL	Available
RIFle	José María Álvarez, Luis Polo	Core, PRD, DTB	Validator	Eclipse Public License	Core+DTB Available, PRD 12/2009
OBR	Gary Hallmark, Oracle	PRD without Import + some of DTB	Producer + Consumer	Proprietary	
IRIS	Adrian Marte, STI Innsbruck	BLD+DTB	Producer + Consumer	TBD open source	End 09
N/A	Stijn Heymans, MichaelKifer	FLD	Core Answer Set Programming Dialect	N/A	available

Fig. 2.21 Implementations of RIF rules (adapted from <http://www.w3.org/2005/rules/wiki/Implementations>)

Eiter et al. (2008) describe the current state of the art of rule integration into the Semantic Web, and addresses open questions and possible future research work in this area. Besides the RIF language, earlier approaches (e.g., Lisi 2008) use other rule languages for the integration of rules into the Semantic Web. Both publications have in common that they show the importance of rule integration for the Semantic Web and describe future challenges like case studies and large(r) scale examples beyond toy examples of semantics for rules plus ontologies, refined studies of computational

(continued)

properties of the various approaches for combining rules and ontologies, and efficient implementations and algorithms for rules plus ontologies.

Zhao and Boley (2008) deal with uncertainty in RIF by extending the RIF language. Boley (2009) defines the central semantics-preserving mappings bridging RIF/XML and RuleML/XML, the mappings between the RIF Presentation Syntax and RIF/XML, and the mappings between RuleML/XML and the Prolog-like RuleML/POSL. Marie (2009) introduces the notion of limited forward compatibility, and describes a low-cost, nondisruptive, extensible implementation. Marie (2009) uses XSLT to specify individual transforms, an XML format to associate them with individual RIF constructs, and the RIF import mechanism to convey the fallback information from RIF producer to RIF consumer. Gordon et al. (2009) describe requirements for rule interchange languages in the legal domain.

2.9.2 *Optimizations for Recursive Rules*

There have been several optimization approaches proposed for recursive rules.

The SNIP approach of McKay and Shapiro (1981) corresponds to the seminaive evaluation in Bancilhon and Ramakrishnan (1986), where recursive rules are reapplied to only the newly inferred facts, rather than to all facts (given and inferred). Top-down evaluation using tables has been proposed in (Tamaki and Sato 1986; Chen and Warren 1996), and static and dynamic filtering in (Kifer and Lozinskii 1990).

The Henschen–Naqvi method (H–N) (Henschen and Naqvi 1984) compiles queries into iterative programs, which are described using relational algebra expressions. Henschen and Naqvi (1984) try to detect “repeated patterns” of relational expressions during the compilation. These patterns provide the basis for the iterative program. Dependence on a repeated pattern severely restricts the applicability of H–N, and it is believed (Bancilhon et al. 1986; Bancilhon and Ramakrishnan 1986) that the application domain of H–N cannot be extended beyond linear axioms. Liu and Stoller (2009) compile rules directly into an implementation in a standard imperative programming language. The generated implementation performs a kind of bottom-up computation based on careful incremental updates with data structure support.

One of the most well-known optimization techniques is the usage of magic sets (Bancilhon et al. 1986; Faber et al. 2005), where a fast computable set is used to early discard irrelevant subgoals. The similar Counting method [described in Bancilhon et al. (1986) and then further developed and generalized by Sacca and Zaniolo (1986) and Beeri and Ramakrishnan

(continued)

(1987)] computes the “distance” from each tuple in a magic set to the tuple of bindings specified in the query. The Counting approach allows more precise selections to be made while computing the query, and this is an advantage over the Magic Sets method. On the other hand, the “distance” between tuples of the magic set and the query binding may not be uniquely defined. In this case, Counting tends to do some superfluous computations compared to Magic Sets.

There are also methods for efficient evaluation of Datalog queries using binary decision diagrams (Lam et al. 2005) and relational databases (Avgustinov et al. 2007).

2.10 Summary and Conclusions

The Semantic Web family of specifications cover many aspects needed for information processing: RDF is defined as simple, but flexible data model, SPARQL as its powerful query language, RDFS and OWL (2) for not only schema information, but also for meta-information considering the meaning of symbols and avoiding redundancies by using inference mechanisms, and RIF as standardized rule language, which is more flexible than just using ontologies.

Chapter 3

External Sorting and B⁺-Trees

Abstract Today’s Semantic Web datasets become increasingly larger containing over one billion triples. The performance of index construction is a crucial factor for the success of large Semantic Web databases. (Large-scale) Indices are typically constructed from externally sorted data. In this chapter, as well as reviewing the data structure B⁺-tree and traditional external sort algorithms, we propose two new external sort approaches: *External chunks-merge sort* and *Distribution Sort for RDF*. The former stores and retrieves chunks from a special chunks heap in order to speed up replacement selection. The latter leverages the RDF-specific properties to construct RDF indices and significantly improves the performance of index construction. Our experimental results show that our approaches significantly speed up RDF index construction and are important techniques for large Semantic Web databases.

3.1 Motivation

Indices are an important data structure for efficient data management and query processing. The invention of the internet, especially of the World Wide Web, revolutionizes the speed and amount of information spread. New data sources keep occurring and are easily obtained. An incremental update to databases will be impractical for insertion of large datasets; instead, indices need to be constructed from scratch. This also applies to many other situations, for example, whenever databases need to be set up from previously made dumps, because of, for example, reconfigurations of the underlying hardware. Therefore, developing efficient techniques to speed up the index construction for large datasets is obviously an important and urgent task for modern databases.

The most widely used index type in databases is the B⁺-tree, which is a variant of the B-tree storing only keys in the interior nodes and all records, that is, the keys and their values, in leaves. B⁺-trees can be built very efficiently from sorted data by avoiding costly node splitting (see (Miller et al. 1977) and adapt its results from B-trees to B⁺-trees). Thus, the performance of index construction from scratch relies heavily on the techniques of data sorting.

Large datasets typically cannot fit into main memory and thus need external sorting, that is, sorting using disk storage. The most well-known efficient external sorting algorithms (Knuth 1998) are *external merge sort* and *distribution sort*.

The external merge sort first generates *initial runs* of sorted data. An initial run is typically computed by reading as much data as possible from input into main memory, and sorting these data using main-memory sorting algorithms. The alternative approach *replacement selection* uses a heap to generate longer initial runs. Runs are written into external storage and merged afterward to larger sorted runs until all the data are sorted.

The distribution sort first distributes the input data according to disjoint intervals into several buckets. The data in these buckets are then sorted using a main-memory sort algorithm if the data fit into main memory; otherwise, the distribution sort is recursively applied.

The main contributions of this chapter include two new approaches to speeding up external sorting for the index construction of large-scale RDF databases:

1. The first approach is a variant of the external merge sort approach. The variant generates longer initial runs (and thus less number of runs) than external merge sort using main-memory sort algorithms, and thus the succeeding merge phase can be processed faster. Since the generation of the initial runs has a similar performance, our approach is faster than external merge sort using main-memory sort algorithms. In comparison to the approach of replacement selection, our approach generates slightly smaller initial runs, and thus the merging phase is slightly slower. However, our approach computes the initial runs much faster since more simple operations are used, such that our approach outperforms replacement selection as well.
2. The second approach is a variant of distribution sort. Our variant considers RDF-specific properties and avoids several unnecessary passes through the RDF data, such that our approach enormously speeds up RDF index construction in comparison to the original distribution sort, and also to the external merge sort algorithms.
3. An experimental evaluation demonstrates the performance improvements of both new approaches.

This chapter contains contributions of (Groppe and Groppe 2010).

3.2 B⁺-trees

B⁺-trees are the most widely used database indices (see Fig. 3.1 for an example of a B⁺-tree). The B⁺-tree is a self-balancing block-oriented search tree. We can store keys and their values in a B⁺-tree and a value can be efficiently retrieved using its key. The nodes in trees without children are called *leaves* and all other nodes are called *interior nodes*. In contrast to B-trees (Bayer and McCreight 1972), in a B⁺-tree all records, that is, the keys and the values (sometimes also pointers to

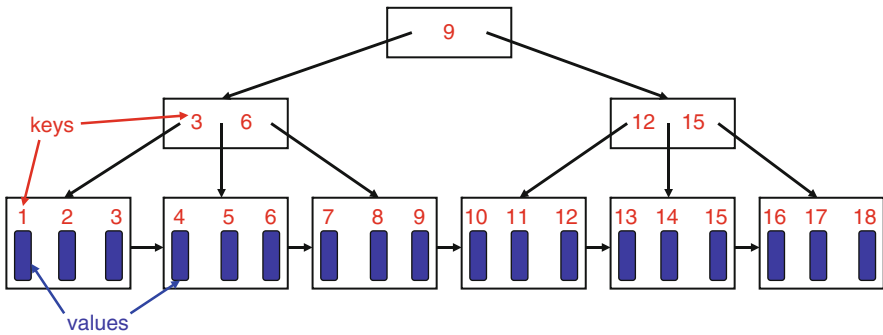


Fig. 3.1 Example of a B⁺-tree

values), are stored in the leaves and the interior nodes contain only keys, such that interior nodes can hold more keys to decrease the height of the overall search tree. The database systems often sort the input data in order to build indices efficiently. Using sorted data, B⁺-trees are constructed without expensive node splitting [see (Miller et al. 1977) and adapt its results from B-trees to B⁺-trees]. Index construction from sorted data plus an extra sorting phase is typically much faster than building the B⁺-tree from unsorted data.

3.2.1 Properties of B⁺-Trees

B⁺-trees have two parameters k and k' . Whereas the parameter k determines the number of keys in an interior node, k' determines the number of key-value pairs in the leaves. Using two parameters k and k' increases the flexibility to react on the higher space consumption of the additional stored values in leaves; that is, k and k' can be balanced such that each interior node and each leaf consumes similar space on disk. The block size is 8 kb in most modern hard disks, such that a space consumption of 8 kb for a node promises best performance. B⁺-trees have the following properties:

- All interior nodes (except of the root) must have at least k and at most $2*k$ keys K_1, \dots, K_j , where $K_1 \leq K_2 \leq \dots \leq K_{j-1} \leq K_j$ holds, and $j+1$ children C_1, \dots, C_{j+1} . The sub-B⁺-tree with root node C_i contains only keys that are larger than K_{i-1} and equal to or less than K_i . For the extreme case C_1 , all keys in the sub-B⁺-tree with root node C_1 must be equal to or less than K_1 , and for C_{j+1} all keys in the sub-B⁺-tree with root node C_{j+1} must be larger than K_j . The interior node is also called the parent (interior) node of its children C_1, \dots, C_{j+1} .
- All leaves (except of the root) contain at least k' and at most $2*k'$ key-value pairs $(k_1, v_1), \dots, (k_j, v_j)$, where k_i represents a key and v_i represents its value, and $k_1 < k_2 < \dots < k_{j-1} < k_j$ holds. Furthermore, all leaves L_1, \dots, L_n are organized in a chain of leaves $L_1 \rightarrow L_2 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$, where all keys of L_i are

smaller than those of L_{i+1} for $1 \leq i < n$. The chain of leaves is realized by a pointer in every leaf (except of the last leaf L_n in the chain), which points to the next leaf in the chain.

- The root node of a B⁺-tree is either a leaf with $0-2*k'$ key-value pairs, if all key-value pairs fit into only one leaf, or is an interior node with at least one key (and two children) and at most $2*k$ keys (and $2*k + 1$ children).
- A sequence C_1, \dots, C_n of B⁺-tree nodes is a *path* with *length* $n - 1$ from the root R to a leaf L , if C_1 is the root R , C_n is the leaf L , and C_i contains the child C_{i+1} for $i \leq 1 < n$. The nodes C_1, \dots, C_{n-1} are also called the *ascendant nodes* of C_n . The length of *all* the paths from the root to its leaves is the same, which is also called the height of a B⁺-tree. Note that this property of B⁺-trees is seldom explicitly stated in literature, but is guaranteed by the insertion and the deletion algorithms of B⁺-trees.

These properties must be guaranteed after the construction of a B⁺-tree from imported data and after each basic operation such as insertion and deletion of a key-value pair. Furthermore, for the basic operations of searching, insertion, and deletion, the maximum number of nodes that must be temporarily stored in main memory for these operations is 3, such that we can guarantee *never* to run out of memory for suitable chosen k and k' .

3.2.2 Self-balancing Property of B⁺-Trees

We will now prove that the height of a B⁺-tree for N stored key-value pairs is in $O(\log(N))$; that is, B⁺-trees are self-balancing.

If the root node is a leaf, then the height of the B⁺-tree is 0.

If the root node is not a leaf, we first determine the maximal height h of a B⁺-tree with N stored key-value pairs. The height of a B⁺-tree for a given number N of key-value pairs becomes maximal if the nodes in a B⁺-tree are “most possible empty”; that is, if all B⁺-tree interior nodes have the smallest possible number of children and the leaves contain the smallest possible number of key-value pairs. This is the case when the root node has two children, each interior node has $k + 1$ children, and the leaves have k' key-value pairs. Thus, the minimal number of key-value pairs stored in such a B⁺-tree with height h is $p_{min} = 2*(k+1)^{h-1}*k'$. As a B⁺-tree with the height h can also store more than p_{min} key-value pairs, $p_{min} \leq N$ holds and thus $h \leq \log_{k+1}(N/(2*k')) + 1$.

We now determine the minimal height h of a B⁺-tree. The height of a B⁺-tree for a given number N of key-value pairs becomes minimal if all nodes in the B⁺-tree are “full up”; that is, all B⁺-tree interior nodes have the largest possible number of children and the leaves contain the largest possible number of key-value pairs as well. This is the case when the root node as well as each interior node has $2*k+1$ children, and the leaves have $2*k'$ key-value pairs. Thus, the maximal number of key-value pairs stored in such a B⁺-tree with height h is $p_{max} = (2*k+1)^h*k'$. As a

B⁺-tree with the height h can also store less than p_{max} key-value pairs, $p_{max} \geq N$ holds and thus $h \geq \log_{2^{*k+1}}(N/k')$.

Therefore, $\log_{2^{*k+1}}(N/k') \leq h \leq \log_{k+1}(N/(2^{*k}k')) + 1$ holds, such that the height of a B⁺-tree is always logarithmic to the number of stored key-value pairs and in this way self-balancing.

For example, for a B⁺-tree with parameters $k = 400$, $k' = 200$ and $N = 4,000,000,000$ (four billions), a B⁺-tree storing N key-value pairs has a height of at most 4.

3.2.3 Searching

Searching for the value of a key k starts at the root node, and searching proceeds in the following way:

If the current node is an interior node with keys K_1, \dots, K_j and children C_1, \dots, C_{j+1} , searching proceeds with the node C_0 if $k \leq K_0$ holds, with the node C_{j+1} if $k > K_j$ holds, and C_i , if $K_{i-1} < k \leq K_i$ holds.

If the current node is a leaf node with key-value pairs $(k_1, v_1), \dots, (k_j, v_j)$, where k_i represents a key and v_i represents its value, then we found the value v_i for the key k if $k = k_i$ with $1 \leq i \leq j$; otherwise, the B⁺-tree does not contain any such key-value pair with key k .

Figure 3.2 presents the accessed path in the B⁺-tree of Fig. 3.1 while searching for the value of the key 8.

3.2.4 Prefix Search in Combination with Sideways Information Passing

While we have described a search for the value of a key in the previous section, we describe the search for all values of keys starting with a given prefix (also called

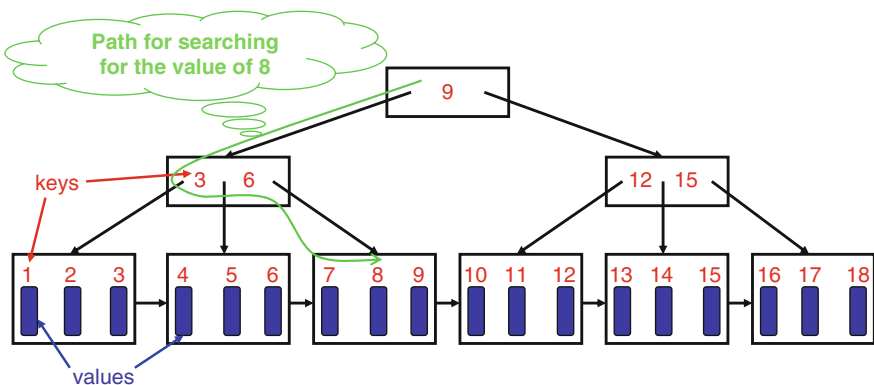


Fig. 3.2 Searching in the B⁺-tree of Fig. 3.1 for the value of the key 8

prefix key) in this section. This kind of search is also called *prefix search*. The applications of prefix searches are widespread. We only name two of them here: If the keys in a B⁺-tree are strings, then the prefix search can be used to retrieve all values of string keys starting with a given prefix key, which can be used, for example, for the wildcard search *dir s** in a command line to display all files starting with *s*. We will also see that prefix search in an index for RDF triples can be used to retrieve the result of a triple pattern.

Indices used to answer Semantic Web queries often index RDF triples according to a certain collation order SPO, SOP, PSO, POS, OSP, or OPS. The collation order determines how the triples are sorted in the index, for example, for the SPO collation order, the triples are primarily sorted according to the subject (S), secondarily according to the predicate (P), and tertiary according to the object (O). If we want to retrieve the result of a triple pattern, where only the subject is not a variable, then we can use this subject as prefix key in the SPO index in order to retrieve all RDF triples matched by the triple pattern. If only the object in a triple pattern is a variable, then we can use the subject and predicate of the triple pattern as prefix key in the SPO index for determining all matching RDF triples. Figure 3.3 contains such a B⁺-tree for RDF data according to the SPO collation order. Note that the B⁺-tree of Fig. 3.3 uses integer ids for RDF literals, which is often used in real SPARQL engines such as RDF3X (Neumann and Weikum 2008) and Hexastore (Weiss et al. 2008).

Like in the normal search as described in the previous subsection, the prefix search starts at the root and recursively searches at the child, which may contain the first key-value pair matched by the prefix key, until the first matching key-value pair in a leaf is found. However, now not the full keys are compared, but just their prefixes with the prefix key (see 1. of Fig. 3.3). Typically the result of a prefix search is returned as iterator, where the single results are returned one by one after calling a *next()* method of the iterator. For the next result, the iterator reads the next key-value pair (*k*, *v*) of the current leaf and returns *v* if *k* conforms to the prefix key (see 2. of Fig. 3.3).

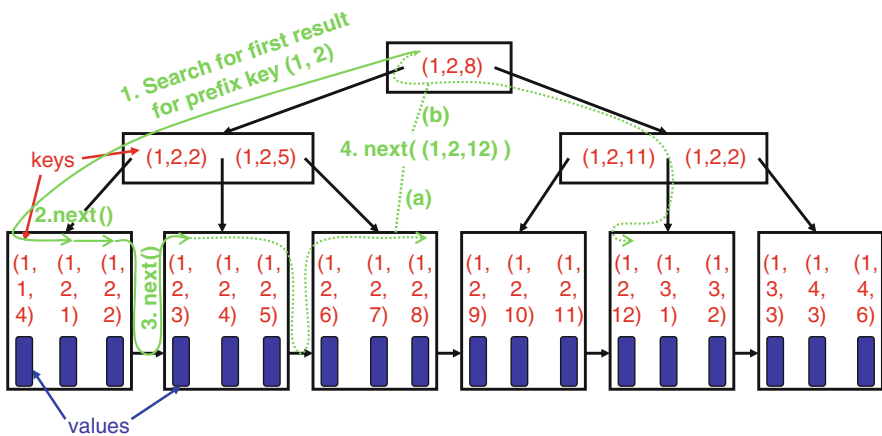


Fig. 3.3 Prefix search using prefix key (1, 2) in a B⁺-tree

Otherwise, if k does not conform to the prefix key, the iterator is closed and does not return any values any more. It may occur that a new leaf must be opened during a $next()$ -call (see 3 of Fig. 3.3) using the pointers in the chain of leaves.

When using sideways information passing strategies for query processing (described in the chapter about physical optimization), a lower limit $lowerLimit$ for the next key of a prefix search is determined during query processing. In order to use this information, the iterator provides a method $next(lowerLimit)$ returning the next result, which is matched by a prefix key and is equal to or larger than $lowerLimit$.

The method $next(lowerLimit)$ can be implemented in two different ways: $next(lowerLimit)$ can just call $next()$ until the determined key-value pair P has a key larger than or equal to $lowerLimit$ (approach *A*). The other way is to use $lowerLimit$ as key and start the search at the root and going over the interior nodes (approach *B*) to reach P . Note that we additionally have to check if P is still matched by the prefix key and not just larger than or equal to $lowerLimit$ when using approach *B*. The approach *B* can be further optimized in the following way: As the key $lowerLimit$ is always larger than the key of the last returned value of the iterator, we can avoid starting the search at the *first* keys in the root and the interior nodes, and can continue the search at the last keys compared during the last search for the prefix key in the root and interior nodes.

If P is far away from the current key-value pair (and this often occurs in large-scale datasets), then we can save much processing time by avoiding going along the chain of leaves as in approach *A*, which possibly has a runtime complexity linear to the number of key-value pairs in the B⁺-tree. When using approach *B*, $next(lowerLimit)$ can jump “directly” to the leaf containing P , and this leads to a logarithmic runtime complexity.

However, if P is in the same (or in the next) leaf, then approach *B* to find P takes longer time than just reading in the key-value pairs along the chain of leaves using approach *A*. Therefore, we need a good strategy to decide when to use approach *A* and when *B*.

In our implementations, we use a heuristic: If we read in a whole leaf without finding P (see 4. Fig. 3.3a for searching the value of a key $\geq (1,2,12)$ in a $next((1, 2,12))$ call), then we use approach *B* (see 4. Fig. 3.3b). Our experiments show that this heuristic yields a good performance.

The algorithm to find all values of keys in a given range $[a, b]$, which is often used for range queries, is very similar to the one for the prefix search. For range queries, the first value is searched with key equal to or larger than a and equal to or smaller than b , and afterward, all succeeding values are returned with keys smaller than b .

3.2.5 Inserting

Indices do not contain only static data. Instead, data are often updated, that is, new data are inserted and old data deleted, to reflect changes in the real world like a new employee. When inserting a new key-value pair into the B⁺-tree, the insertion

position in a leaf is first searched for using a search algorithm similar to the one for searching the value of a key. If we inserted the new key-value pair as last key-value pair in the leaf (see Figs. 3.4 and 3.5 for an example), then we may have to update

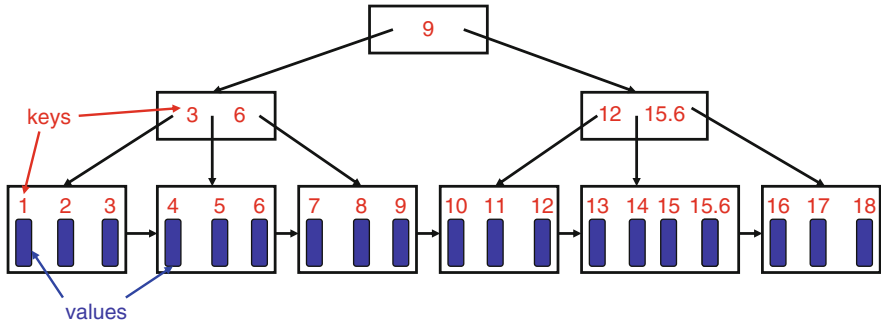


Fig. 3.4 Inserting 15.6 in the B⁺-tree (with $k=1$ and $k'=2$) of Fig. 3.1

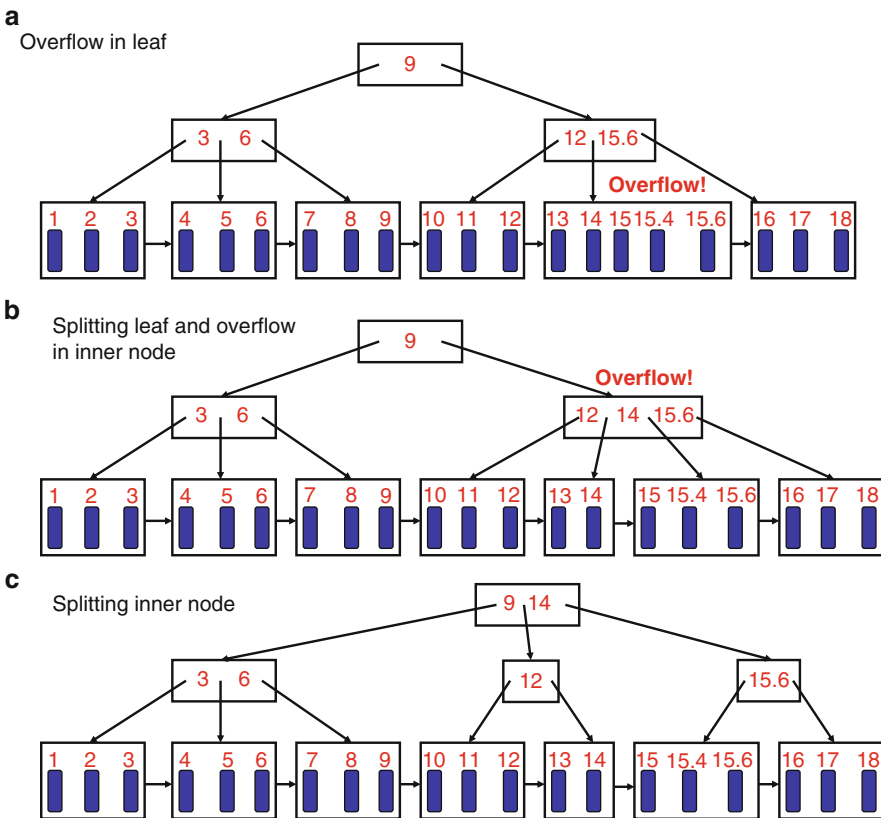


Fig. 3.5 Inserting 15.4 in the B⁺-tree (with $k=1$ and $k'=2$) of Fig. 3.4

an/several interior node(s) ascendant to the leaf with this new largest key of the leaf, as interior nodes contain the largest keys of their subtrees.

The properties of the B⁺-tree do not allow that an updated leaf contains more than $2*k'$ key-value pairs, which is called an *overflow*. This can be resolved by splitting the leaf with an overflow into two leaves: the first leaf takes over the first k' key-value pairs and the second leaf takes over the remaining key-value pairs [see Fig. 3.5a]. We also need to update the interior node containing the old leaf as child: The last keys of the new two leaves replace the key of the old leaf and the old leaf is replaced by the new two leaves in the list of children.

Again, the updated interior node can have an overflow with more than $2*k$ key and $2*k+1$ children. Therefore, splitting interior nodes and updating their parents (see Fig. 3.5b and c) may need to be performed several times. In the worst case, the process must be repeated until to the root node.

Using this algorithm, the properties of the B⁺-tree can be guaranteed. However, we can try to avoid splitting a node, which introduces a new level in the B⁺-tree in the worst case and therefore slows down operations in the B⁺-tree, in the following way: We assume N to be a node with an overflow. One of the neighbour nodes of N , that is, the nodes left and right to N at the same level in the B⁺-tree, may have less than $2*k'$ key-value pairs in the case of leaves, or less than $2*k$ keys and $2*k + 1$ children in the case of interior nodes. Then we can avoid splitting N by shifting a key-value pair in the case of leaves or a child in the case of interior nodes from N to this neighbour node. Note that we may have to update the keys in the ascendant interior nodes of N and of the neighbour nodes, but we do not have to add keys and children in the ascendant interior nodes.

3.2.6 Deleting

In order to delete a key-value pair from a leaf, we search for the key-value pair in the B⁺-tree (see Fig. 3.6). Note that we do not need to (but could) update the keys in ascendant interior nodes (e.g., 9 in the root in a) of Fig. 3.6) for simple deletions, as all searches are still successful even if the old keys remain in the interior nodes. Deletion of a key-value pair can cause an *underflow*, that is, a leaf contains less than k' key-value pairs (see Fig. 3.6b, d, f), which is not allowed in a B⁺-tree according to its properties.

In this case, if one neighbour of the leaf has more than k' key-value pairs, then we can shift the first (in the case of a right neighbour) or last (in the case of a left neighbour) key-value pair from it to the leaf with an underflow in order to recover the B⁺-tree properties (see Fig. 3.6c and e). Furthermore, we have to update the keys in the ascendant interior nodes of the leaves correspondingly (see Fig. 3.6c and e).

If a neighbour node has exactly k' key-value pairs, then we can merge the two leaves to one leaf, which will have $2*k' - 1$ key-value pairs (Fig. 3.6g). One child in an interior node is deleted when two nodes are merged, which can cause again an underflow in this interior node. Therefore, deleting a key-value pair might cause the

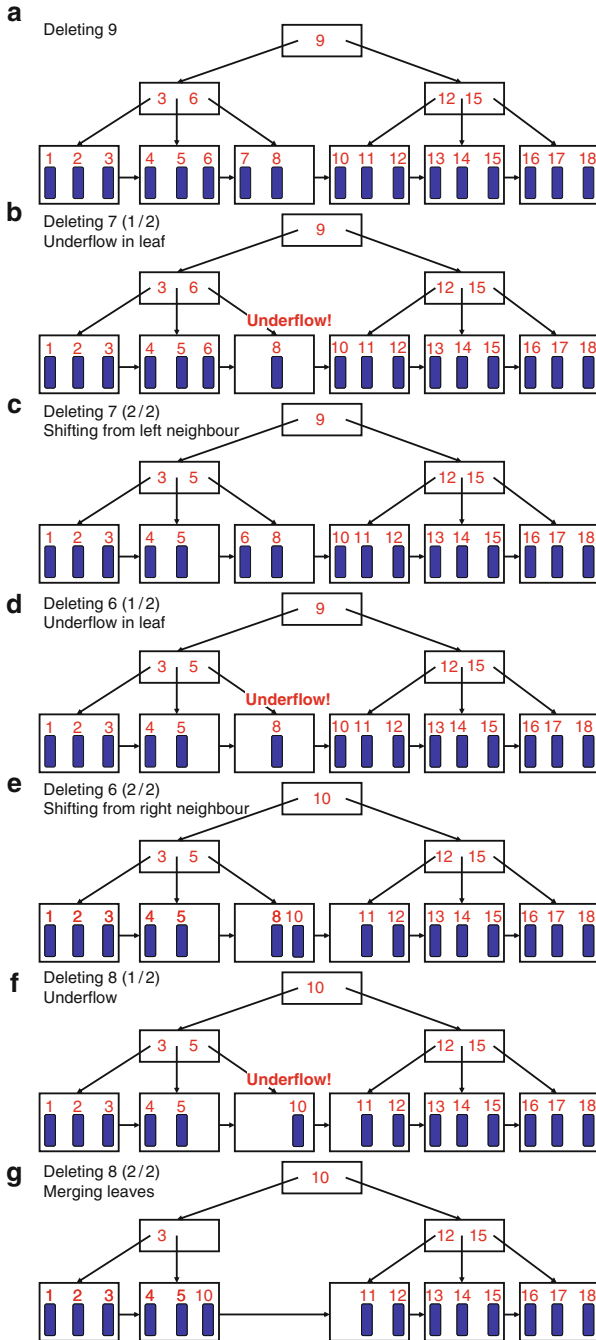


Fig. 3.6 Deleting in the B⁺-tree (with $k=1$ and $k'=2$) of Fig. 3.1

actions shifting a key-value pair from a neighbour or merging two nodes performed many times. In the worst case, we have to merge interior nodes until the root node, and even may have to replace the root with a merged node, such that the height of the B⁺-tree is reduced.

3.2.7 B⁺-Tree Construction from a large Dataset

If we have to construct a new B⁺-tree from a large set with N key-value pairs, it is wise to first sort the key-value pairs according to the keys. Afterward, we can construct the B⁺-tree in one pass through the sorted data (see Fig. 3.7 for an example): we first determine the number d of leaves by calculating $\lceil N/2^*k' \rceil$. The number p of their parent interior nodes can be also determined by calculating $\lceil d/2^*k \rceil$. We can proceed to calculate the number of interior nodes at the level above in the B⁺-tree by computing $\lceil p/2^*k \rceil$ and so on until the number of nodes are computed for all levels of the B⁺-tree.

With the number of nodes at each level of the B⁺-tree computed, we can equally distribute the key-value pairs at the leaves and the corresponding keys and children at the interior nodes, such that we neither have an underflow nor an overflow at any node in each level. Finally, we create the leaves and interior nodes by one pass through the sorted data.

Whenever a leaf L is finished, we have to add it as child in its parent interior node at the upper level with the corresponding key (i.e., the maximum key in L). If the parent node does not exist, we create one for it. A finished interior node I is then recursively added as a child of a node at its upper level with the corresponding key (i.e., the maximum key in the subtree with root I).

For each level of the B⁺-tree, at most one node has not been finished at the same time during applying this algorithm. Therefore, the height of the B⁺-tree determines the number of file output streams to be used at the same time to store unfinished nodes, which is no problem even for large datasets. In our experiments, the algorithm does not run out of memory for any dataset even not for the very large ones with over 1.5 billion triples.

3.3 Heap

A (min-) heap is an efficient data structure to retrieve the smallest item from the items stored in the heap [see (Williams 1964)]. Adding an item into the heap and removing the smallest item from the heap is done in $O(\log n)$, where n is the size of the heap. Internally, the heap is organized as tree, most often as complete binary tree. The root of each subtree contains the smallest item of the subtree. Complete binary trees can be memory efficiently stored in arrays, where the index of the children and the parent can be computed by simple formulas. When the smallest

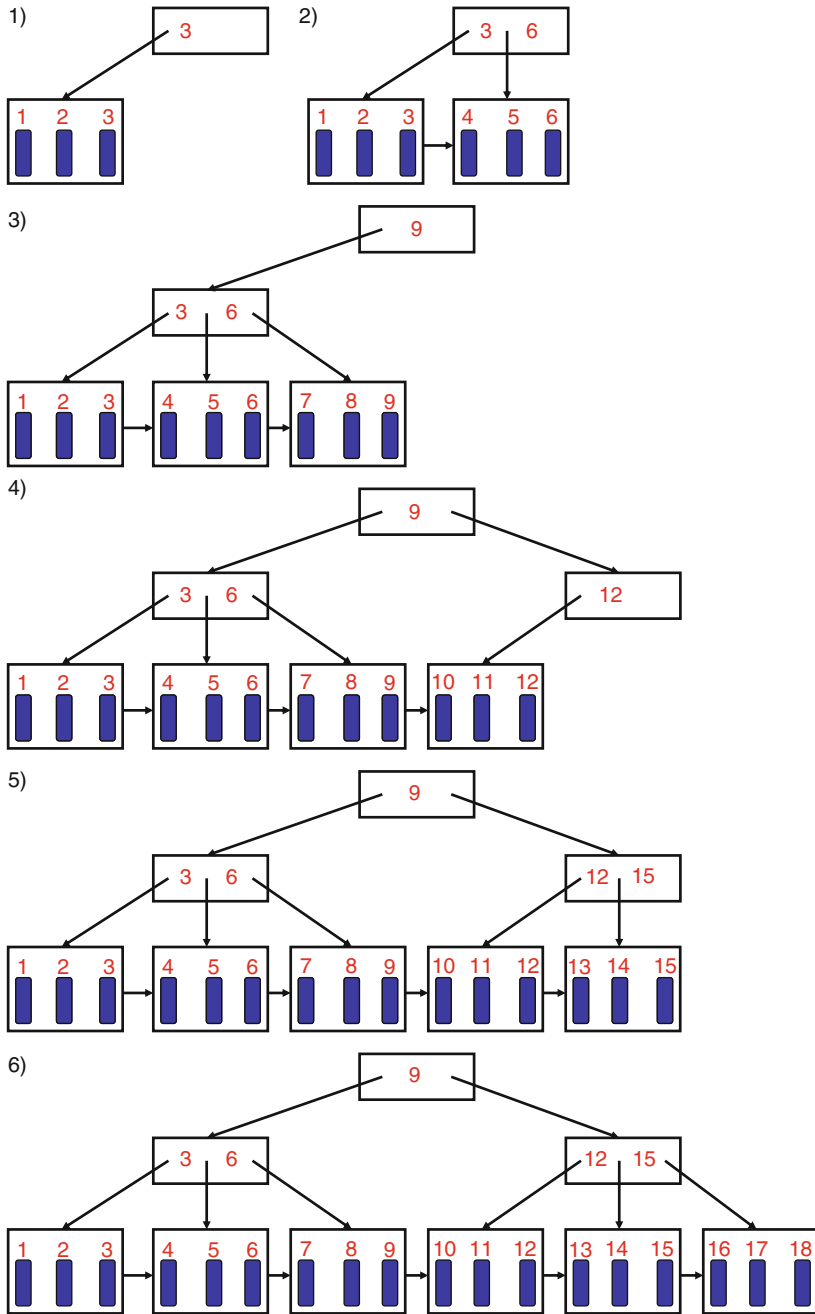


Fig. 3.7 Constructing B⁺-tree in one pass from sorted data, where the B⁺-tree is presented after each constructed leaf

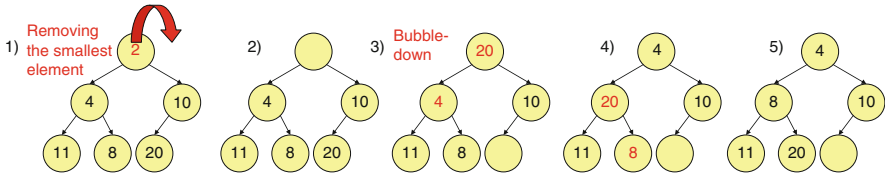


Fig. 3.8 Removing an element from a heap

Fig. 3.9 Inserting a new element into the heap

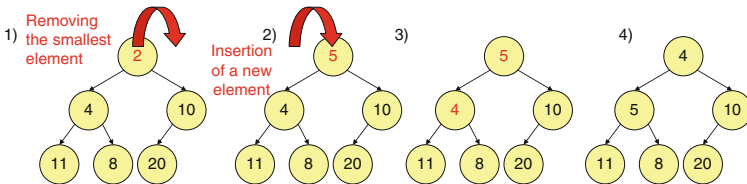
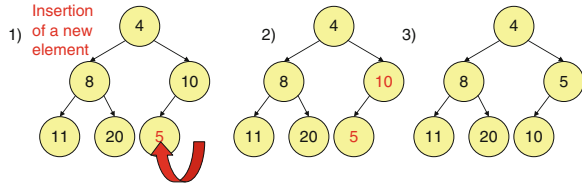


Fig. 3.10 Optimizing removing and insertion of a new element into the heap

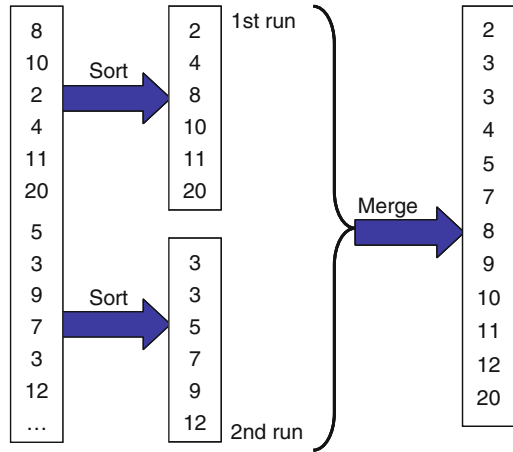
item is taken away, an item in a leaf is moved to the root. Afterward, the root item is recursively swapped with its minimum child if the minimum child is smaller than it (*bubble-down*). For an example of removing the smallest item from a heap, see Fig. 3.8. Adding an item is done by inserting the item in an empty leaf to the heap tree and swapping the item with its parent as long as it is smaller than its parent (*bubble-up*). See Fig. 3.9 for an example of inserting. Therefore, the smallest item in the heap is always stored in the root of the tree.

After the smallest element in the heap is taken away, a succeeding insertion of a new element can be optimized by first placing the new element in the root and then performing a bubble-down operation. This approach to optimize a pair of removing and insertion operations avoids the bubble-up operation (see Fig. 3.10).

3.4 (External) Merge Sort

(External) merge sort [see (Knuth 1998)] is known to be one of the best sorting algorithms for external sorting, that is, if the data are too large to fit into the main memory. The merge sort algorithm first generates several initial *runs* from the given data. Each run consists of a sorted subset of the given data. Several initial runs are afterward merged to generate a new round of runs. Instead of merging only two

Fig. 3.11 External merge sort



runs, it is more efficient to merge several runs. In order to generate a new run from several runs, we need always to find the smallest items from these runs. Thus, a heap is the ideal data structure to perform this task. For an example of the application of external merge sort, see Fig. 3.11. This process is repeated until all the data are sorted.

The runs can be generated by reading as much data into main memory as possible, sorting this data, and write this run to external storage. For sorting the data in main memory, any *main memory sort* algorithm can be chosen (Knuth 1998), for example, quicksort, (main-memory) merge sort (and its parallel version), or heap sort, which are well known to be very fast.

3.5 Replacement Selection

Another approach, called *replacement selection* (Friend 1956), uses a heap to increase the length of the initial runs on average by a factor of 2. Whenever the heap is full, its root item is retrieved and written to the current run. If an item is inserted into the heap it is checked if it can be still written into the current run, that is, if it is greater than or equal to the last item written into the current run. In this case, the run number of the current run is attached to this item. If the item cannot be written into the current run (as it is smaller than the last item written into the current run), the run number of the next run, which is the run number of the current run plus 1, is attached to the item. The items with a higher run number are regarded to be larger in the heap, that is, the run number is the primary order criterion and the item is the second-order criterion in the heap, such that first all items of the current run are stored. If the root item in the heap is smaller than the last item of the current run, the current run is closed, and a new run is created and becomes the current run. We present the first eight steps of an example for replacement selection in Fig. 3.12.

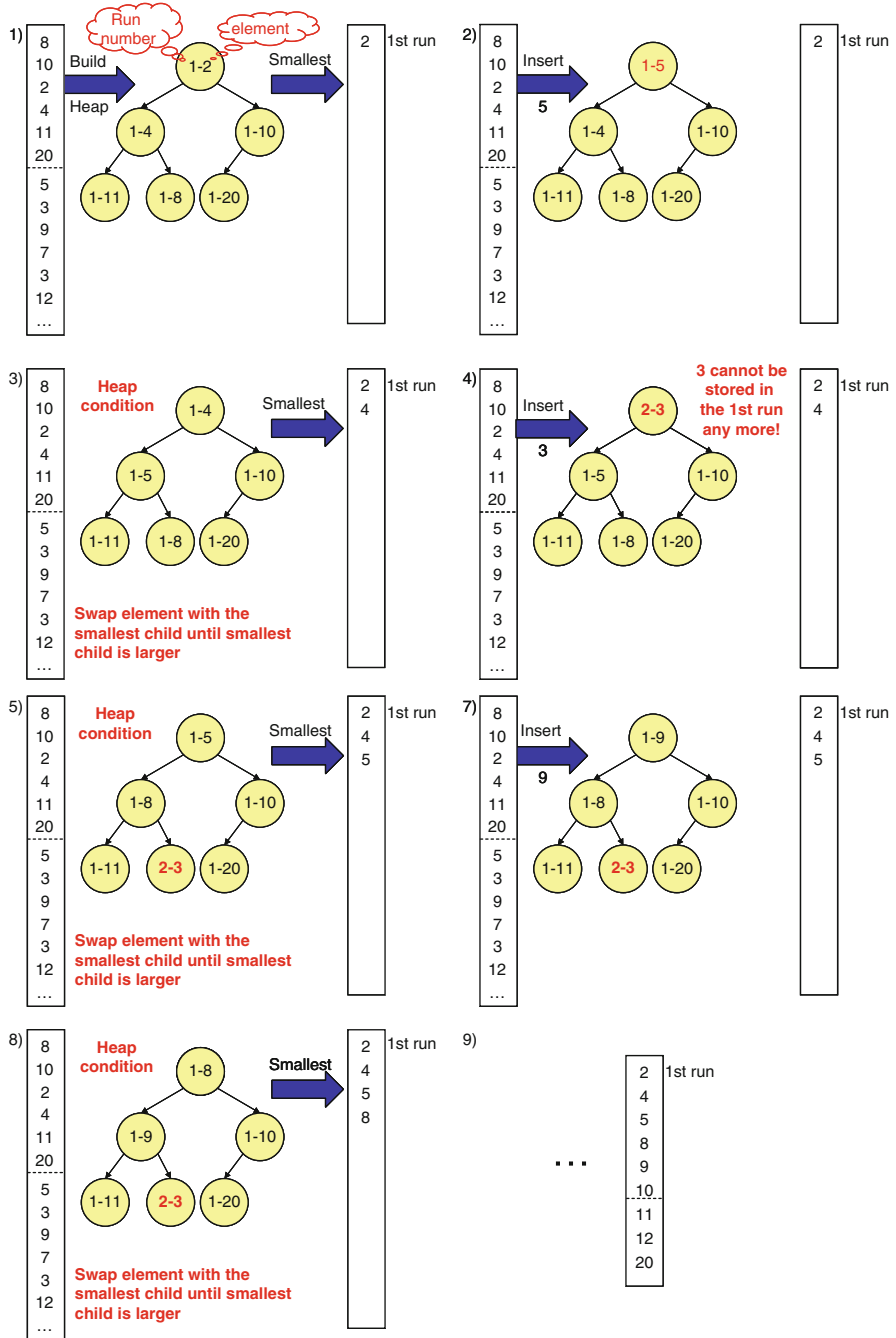


Fig. 3.12 Example for replacement selection

Figure 3.12 also contains the first run of the example (see 9 of Fig. 3.12), which is three items larger than the first run of the external merge sort approach in Fig. 3.11.

3.6 External Chunks Merge Sort

Generating the initial runs using main-memory sort algorithms is very fast, but produces a high number of initial runs, and thus the following merge phase is slow. In contrast, replacement selection produces a smaller number of initial runs, and thus there is a faster merge phase. However, maintaining a heap is time-consuming, as more operations are needed in comparison to using main-memory sort algorithms for the initial runs.

If we can take the advantages of these two approaches: the speed of main-memory sort and the larger runs of replacement selection, then we will be able to improve the sorting performance, and thus speeding up index construction. Therefore, we propose a new and efficient data structure, *k-chunks heap*. A *k-chunks heap* is a block-oriented variant of the heap, where usually blocks of k items are added to and retrieved from the heap. The *k-chunks heap* uses only very simple and thus fast operations for blocks of k items. The application of our *k-chunks heaps* to external merge sort speeds up the computation of the initial runs in comparison to replacement selection and generates the larger initial runs (and thus a smaller number of initial runs) in comparison to using main-memory sort algorithms. The following merge phase is also hence fast processed.

Note that rather than organized as tree, our *k-chunks heap* holds a sorted list of items (see Fig. 3.13). Whenever a chunk of k items is added, the k items are sorted using an efficient main-memory sort algorithm. Afterward, the sorted k items are merged with the items (which are already sorted) in the heap. If a chunk of the k smallest items is requested, then the first k items of the heap are returned, and the remaining heap content remains sorted. Note that these operations can be memory efficiently performed by using arrays instead of lists of items.

Applying our *k-chunks heap* to the external merge sort, we create a variant of the external merge sort, which we call *external chunks merge sort*. Our approach first reads as much data into main memory as possible, sorts the data, and stores it into a *k-chunks heap*. Instead of retrieving one item from the heap and storing it into the current run, the first k items of the heap are retrieved and written into the current run. Afterward, k new items are read in and added to the heap. When we read the k new items, we also check if they can be put into the current run. If an item cannot be written into the current run (i.e., it is smaller than the last item of the current run), it is attached to the number of the next initial run (i.e., a number, which is larger than the number of the current run). Otherwise, the item is assigned with the number of the current run. When we sort these k items, the items with smaller run number are ordered before those with larger ones.

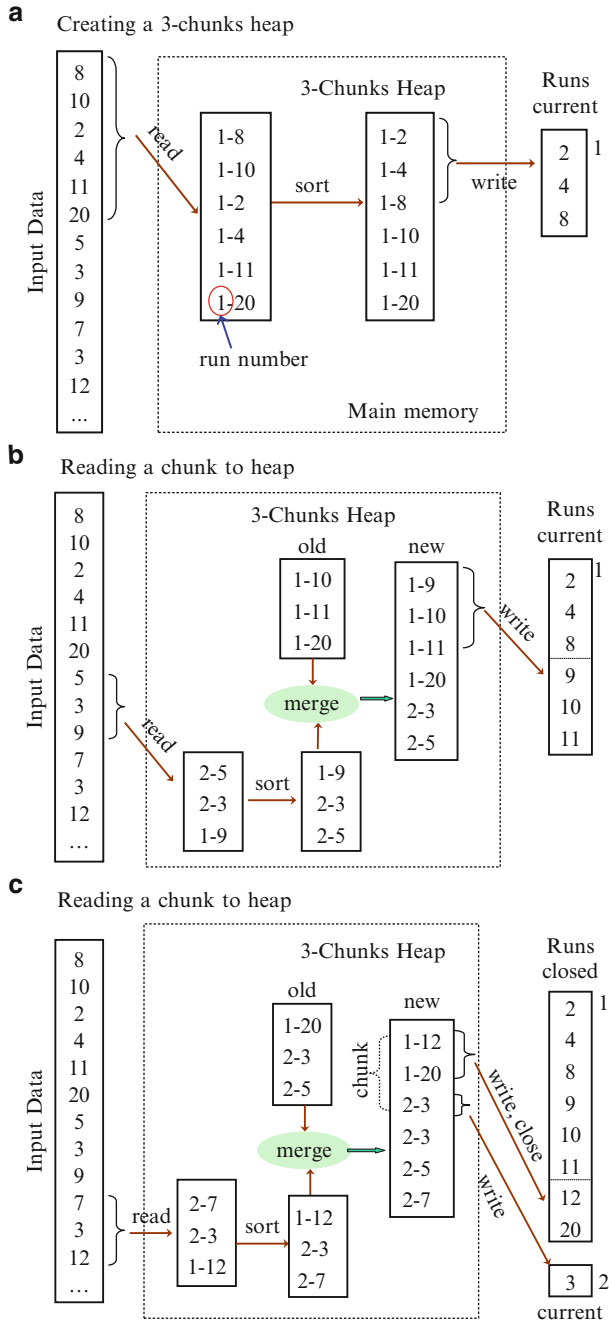


Fig. 3.13 External chunks-merge sort

Afterward, the sorted new items are merged with the (old) heap content, according to the run number as the primary comparison criterion and the value of the item as the secondary comparison criterion. Merging in this way guarantees that all the items in the heap, which will be written into the current run, are ordered before the items, which will be stored into the next run. Figure 3.13 demonstrates our external chunks-merge sort approach. Afterward, we once again take the first k items from the heap. If all k items are attached to the number of the current run, they are written into the current run. If only a part of the k items are attached to the number of the current run, this part is written into the current run, and then the current run is closed. A new run is opened and becomes the current run, and the remaining of the retrieved k items is written into it.

After all input data are read and stored in initial runs, our external merge sort algorithm merges the initial runs like in the traditional external merge sort algorithms.

The external chunks merge sort generates fewer initial runs than the main-memory sort, but more than replacement selection. However, our external chunks merge sort approach uses simpler (and thus faster) operations than replacement selection. Therefore, our approach is more efficient than replacement selection. We demonstrate the efficiency by our performance study. Our experimental results show that our external chunks-merge sort approach outperforms replacement selection and main-memory sort algorithms greatly.

3.7 Distribution Sort

Another well-known external sorting algorithm is distribution sort (Knuth 1998). Distribution sort first retrieves k values v_1, \dots, v_k (*distribution keys*) from the input, where $v_1 < v_2 < \dots < v_k$ according to an order relation $<$. Afterward, distribution sort distributes the input data according to v_1, \dots, v_k into $k+1$ different buckets: the data, which are less than or equal to v_1 are stored in the first bucket, the data larger than v_1 and less than or equal to v_2 are stored in the second bucket, and so on. The choice of v_1, \dots, v_k is critical for the performance of the algorithm, as the approach is fastest for buckets with similar sizes. A fast and simple approach is choosing v_1, \dots, v_k randomly from the input data. More complex approaches (see e.g., Nodine and Vitter 1993) use histograms to improve the choice of the distribution keys.

Afterward, every bucket is sorted by a main-memory sorting algorithm if the whole bucket fits into main memory; otherwise, distribution sort is recursively applied to this bucket. The overall sorted sequence is the concatenation of the sorted buckets. For an example of distribution sort, see Fig. 3.14.

Using distribution sort, the merging phase can be avoided. The distribution phase is usually performed faster than the merging phase. Therefore, distribution sort earns considerable advantages in comparison to external merge sorting algorithms.

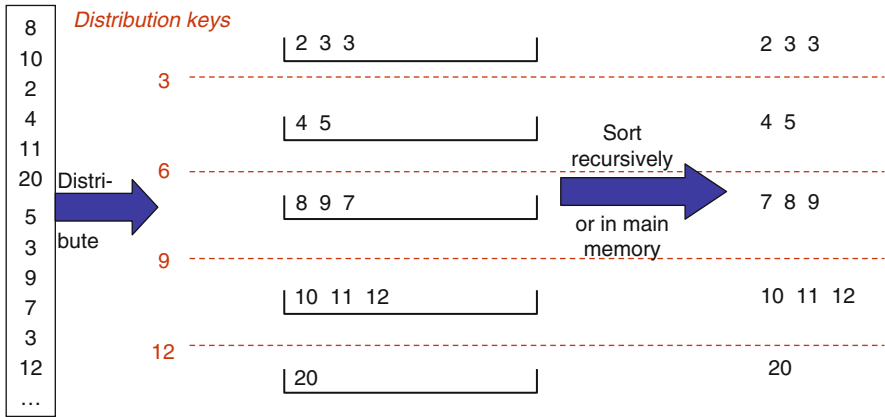


Fig. 3.14 Example for distribution sort

3.8 RDF Distribution Sort

The most efficient indices for managing and accessing RDF data are the six indices from the six collation orders SPO, SOP, PSO, POS, OSP, and OPS (see e.g., Neumann and Weikum 2008; Weiss et al. 2008). Therefore, we need to sort the RDF data six times according the six collation orders in order to construct RDF indices efficiently.

Applying the general-purpose distribution sort for RDF index construction has two main disadvantages:

- The input data are read 12 times (six times for retrieving the distribution keys and six times for distribution of each of the collation orders SPO, SOP, PSO, POS, OSP, and OPS).
- For each distribution, full triples consisting of three components (subject, predicate, and object) are compared.

These disadvantages significantly impact the performance of index construction, especially for large RDF databases. Therefore, we propose two new variants of distribution sort in this chapter to avoid these severe shortcomings.

We define $<_c$ to be an order relation between triples according to a collation order $c \in \{SPO, SOP, PSO, POS, OSP, OPS\}$; for example, for two triples (s_1, p_1, o_1) and (s_2, p_2, o_2) , the order relation $<_{SPO}$ holds; that is, $(s_1, p_1, o_1) <_{SPO} (s_2, p_2, o_2)$, if $s_1 < s_2$ or $(s_1 = s_2$ and $(p_1 < p_2$ or $(p_1 = p_2$ and $o_1 < o_2)))$.

The comparison between the components $s_1, s_2, p_1, p_2, o_1,$ and o_2 are defined based on the order of blank nodes, uris/iris, and literals as specified in the SPARQL specification (Prud'hommeaux and Seaborne 2008). This allows plan generators of SPARQL evaluators to consider the plans, which use the RDF indices to directly

retrieve sorted data according to sort criteria specified in a SPARQL query, and thus to avoid extra sorting phases at runtime.

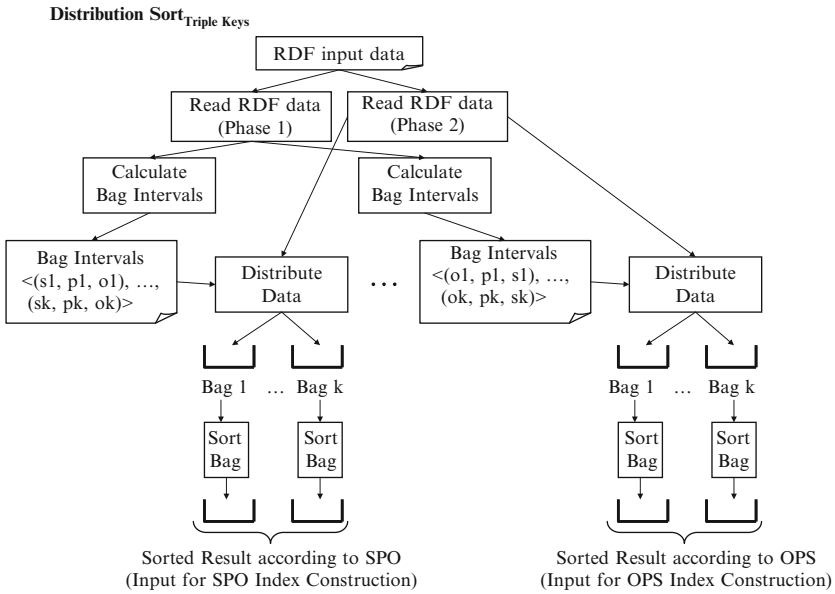
If a SPARQL engine maps RDF literals to integer ids using a dictionary, then the comparison between the components $s_1, s_2, p_1, p_2, o_1,$ and o_2 are usually defined based on the order of the integer ids in all operations except of sorting. As consequence, sort operations cannot be eliminated, but other operations can be optimized like the application of merge joins using this order and optimization of duplicate elimination in already sorted query results.

Our first variant is *Distribution Sort Triple Keys* (see Fig. 3.15a). Using this approach, the input data need to be read only twice: one time for retrieving the distribution keys (for the different indices) and one time for distribution of the data (see Fig. 3.15a). We first randomly retrieve k triples $v_{c,1}, \dots, v_{c,k}$ (*distribution keys*) by using only one pass through the input for every of the six collation orders $c \in \{\text{SPO, SOP, PSO, POS, OSP, OPS}\}$, where $v_{c,1} <_c v_{c,2} <_c \dots <_c v_{c,k}$. In another pass through the data, the triples are distributed to $k+1$ bags for each collation order, where the distribution keys play the role of the bag borders.

In more detail, for each read triple t and each collation order c , we search for the distribution key $v_{c,i}$, where $v_{c,i-1} <_c t \leq v_{c,i}$, and store the triple t in the bag i of the collation order c . Note that we have to consider the two special cases of the first and the last bag: If $t \leq v_{c,1}$, then we store t in the first bag, if $v_{c,k} <_c t$, then we store t in the last bag $k + 1$ of the collation order c . We then sort each bag in main memory, if the bag content fits into main memory; otherwise, we recursively apply *Distribution Sort Triple Keys* to the bag. The overall sorted sequence of a collation order c is the concatenation of the sorted buckets of c .

The second variant is *Distribution Sort Component Keys* (see Fig. 3.15b). Similar to *Distribution Sort Triple Keys*, the input is read twice: one time for randomly choosing the distribution keys and one time for distributing the data. Instead of distributing the data six times, this variant distributes the data only three times and considers only the bags for the subjects, predicates, and objects of the RDF triples. Thus, the distribution keys are chosen only from the subjects of the triples for the subject-distributed bags, from the predicates for the predicate-distributed bags and from objects for the object-distributed bags. The bags, containing distributed data according to the subjects, can be used to sort the data for the SPO and SOP collation orders; the predicate-distributed bags for the PSO and POS collation orders; and the object-distributed bags for the OSP and OPS collation orders. As well as only distributing the RDF data three times instead of six times, the variant also avoids the comparison of full triples, and just compares one component (subject, predicate, or object) of a triple. Comparing only one component rather than three components of triples can significantly save processing time for large datasets. Our experiments show that the distribution of the input data using this approach is similar as well as the approach *Distribution Sort Triple Keys*. *Distribution Sort Component Keys* significantly improves the performance of index construction by reducing the number of passes through data, of distribution, and of comparison operations.

a



b

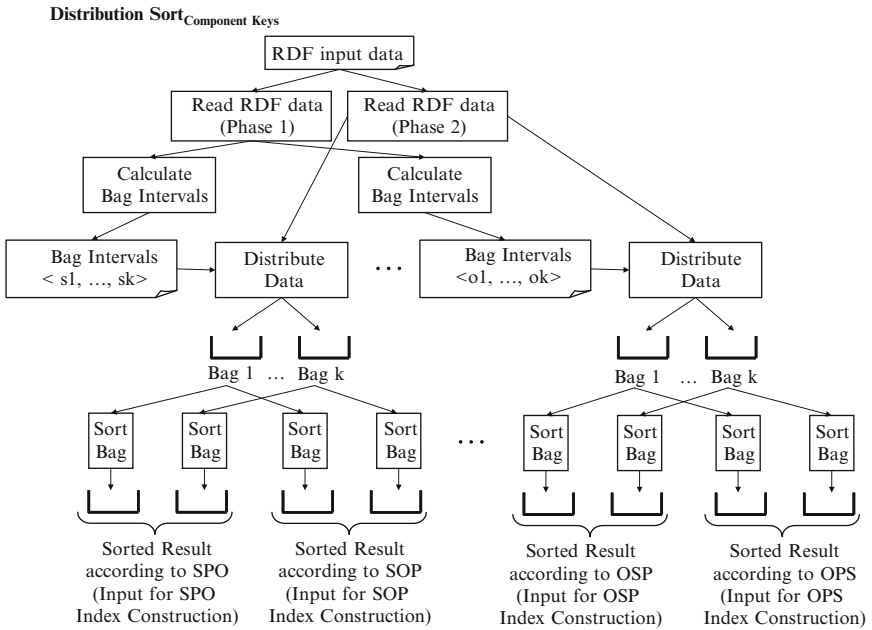


Fig. 3.15 (a) Distribution sort_{Triple Keys} and (b) Distribution Sort_{Component Keys}

3.9 Experimental Analysis

In the performance study, we use real-world data, and synthetic data, as synthetic data with different sizes can be generated.

In our experiments, we first sort the data in parallel according to the 6 collation orders SPO, SOP, PSO, POS, OSP, and OPS. Afterward, we generate in parallel six B⁺-trees for keys from the six sorted datasets. We use a prefix codemap for the prefixes of uris and iris in order to significantly reduce the storage space consumption.

We compare several different approaches for computing the initial runs: the most important standard approaches (the external merge sort using replacement selection, and using main-memory quicksort/parallel merge sort), and our new approaches (External Chunks-Merge Sort, Distribution Sort_{Triple Keys}, and Distribution Sort_{Component Keys}). For replacement selection, we use an optimized heap, which avoids a bubble-up operation in the two succeeding operations retrieving the smallest item of the heap and adding a new item to the heap: After the root item is taken away, instead of the standard operations (i.e., moving the last leaf node to the root, performing a bubble-down operation, adding the new item as the last leaf node and performing a bubble-up operation), we directly insert the new item at the root, and perform just one single bubble-down operation. This optimization significantly speeds up the replacement selection by avoiding a bubble-up operation. Distribution Sort_{Component Keys} uses Distribution Sort_{Triple Keys} when bags are too big to be sorted in main memory and must be further distributed.

In the experiments, we have used 100 bags and 1,000 bags in Distribution Sort_{Triple Keys} and Distribution Sort_{Component Keys}. We have used an upper bound of 2^{11} , 2^{13} , and 2^{15} items in main memory for external merge sorts using main-memory quicksort/parallel merge sort and external chunks-merge sorts. The replacement selection variants have an upper bound of $2^{11}-1$, $2^{13}-1$, and $2^{15}-1$ items in main memory (which are the space requirements of corresponding complete binary trees). We have used these upper bounds for the items in main memory because the sort algorithms are the fastest for these upper bounds. Although using larger upper bounds lead to less initial runs (and thus the merge phase is faster processed), the main-memory sort phases as well as heap operations for the generation of the initial runs are much slower for larger upper bounds. Thus, the main-memory sort approaches and replacement selection perform best for certain upper bounds (neither smaller nor larger upper bounds). The external chunks-merge sorts are performed with the chunks of half of the items hold in main memory (i.e., $(2^{11}/2)$ -chunks-merge sort, $(2^{13}/2)$ -chunks merge sort, and $(2^{15}/2)$ -chunks merge sort), and 1/3 of the items hold in main memory (i.e., $(2^{11}/3)$ -chunks merge sort, $(2^{13}/3)$ -chunks merge sort, and $(2^{15}/3)$ -chunks merge sort). Note that for larger chunks, the time used for sorting increases, but the number of chunks is less, and thus the number of merging phases is less. Therefore, there exists also an optimal chunk size, which is performed the best.

In our figures, we present the average of five execution times of reading input, sorting, and generating indices.

3.9.1 *SP²B Dataset*

The SP²B benchmark (Schmidt et al. 2009) includes a set of 18 queries, which contain more features of SPARQL and address more optimization techniques than many other Semantic Web benchmarks such as LUBM benchmark (Guo et al. 2005). The SP²B benchmark uses a data generator, which can generate data of different sizes. In our experiments, we have used datasets with one million and ten million triples. The SP²B datasets imitate an RDF version of the real-world DBLP dataset (Ley 2010); that is, the data structure of the SP²B datasets is very similar to real-world data.

The test system for this dataset uses an Intel Core 2 Quad CPU Q9400 with 2.66 GHz, 4 GB main memory, Windows XP Professional (32 bit) and Java 1.6.

We have used the same legend presented in Fig. 3.16 for all figures of the SP²B dataset.

The total times for reading data, sorting, and index construction (see Figs. 3.17 and 3.18) show that

- External merge sort using parallel merge sort for the initial runs is faster than external merge sort using quicksort and replacement selection,
- Chunks-merge sort is significantly faster than the other external merge sort approaches and replacement selection,
- Distribution Sort _{Triple Keys} is competitive with the other external merge sort approaches and replacement selection, and
- Distribution Sort _{Component Keys} has much less main-memory requirements and outperforms all other external sorting approaches significantly.

Let us first have a look at the external merge sort and replacement selection variants. The generation of initial runs (see Figs. 3.19 and 3.20) is similar fast when using quicksort and replacement selection, and is very fast when using parallel merge sort because of the efficient main-memory sort algorithm. However, the chunks-merge sorts generate the initial runs fastest, since only smaller chunks need to be sorted, and merging the new sorted chunk with the old heap content can be done in parallel with reading in the next chunk.

Replacement selection performs best (especially for the larger datasets) in the merging phase (see Figs. 3.21 and 3.22); the external merge sort approaches using main-memory sort algorithms perform worst; the external chunks-merge sorts are between these two approaches. It is obvious that the times used for merging depend on the total number of runs (see Figs. 3.21, 3.22, and 3.23).

Looking at the distribution sort variants, we see that using 1,000 instead of 100 bags significantly reduces the number of distribution passes (see Fig. 3.24), and Distribution Sort _{Triple Keys} and Distribution Sort _{Component Keys} have a similar number of distribution phases. In the first distribution round, Distribution Sort _{Component Keys} distributes the data only to 3 instead of 6 bags, each of which is used for two different collation orders. Furthermore, only one component, rather

- | | |
|--------------------------|---|
| <input type="checkbox"/> | External Merge Sort using Quicksort on 2 ¹¹ items |
| <input type="checkbox"/> | External Merge Sort using Parallel Mergesort on 2 ¹¹ items |
| <input type="checkbox"/> | Replacement Selection using Heap with 2 ¹¹ -1 items |
| <input type="checkbox"/> | External (2 ¹¹ /2)-Chunks Merge Sort |
| <input type="checkbox"/> | External (2 ¹¹ /3)-Chunks Merge Sort |
| <input type="checkbox"/> | External Merge Sort using Quicksort on 2 ¹³ items |
| <input type="checkbox"/> | External Merge Sort using Parallel Mergesort on 2 ¹³ items |
| <input type="checkbox"/> | Replacement Selection using Heap with 2 ¹³ -1 items |
| <input type="checkbox"/> | External (2 ¹³ /2)-Chunks Merge Sort |
| <input type="checkbox"/> | External (2 ¹³ /3)-Chunks Merge Sort |
| <input type="checkbox"/> | External Merge Sort using Quicksort on 2 ¹⁵ items |
| <input type="checkbox"/> | External Merge Sort using Parallel Mergesort on 2 ¹⁵ items |
| <input type="checkbox"/> | Replacement Selection using Heap with 2 ¹⁵ -1 items |
| <input type="checkbox"/> | External (2 ¹⁵ /2)-Chunks Merge Sort |
| <input type="checkbox"/> | External (2 ¹⁵ /3)-Chunks Merge Sort |
| <input type="checkbox"/> | Distribution Sort Triple Keys 100 Bags |
| <input type="checkbox"/> | Distribution Sort Component Keys 100 Bags |
| <input type="checkbox"/> | Distribution Sort Triple Keys 1000 Bags |
| <input type="checkbox"/> | Distribution Sort Component Keys 1000 Bags |

Fig. 3.16 Legend used in the figures of the SP²B experiments

than all the components of triples, is compared. All these factors significantly speed up sorting the data for RDF index construction.

3.9.2 *Yago Dataset*

The Yago dataset (Suchanek et al. 2007) consists of facts extracted from the info boxes and category system of Wikipedia and are integrated with the WordNet thesaurus. Thus, this dataset is more homogeneous compared with, for example,

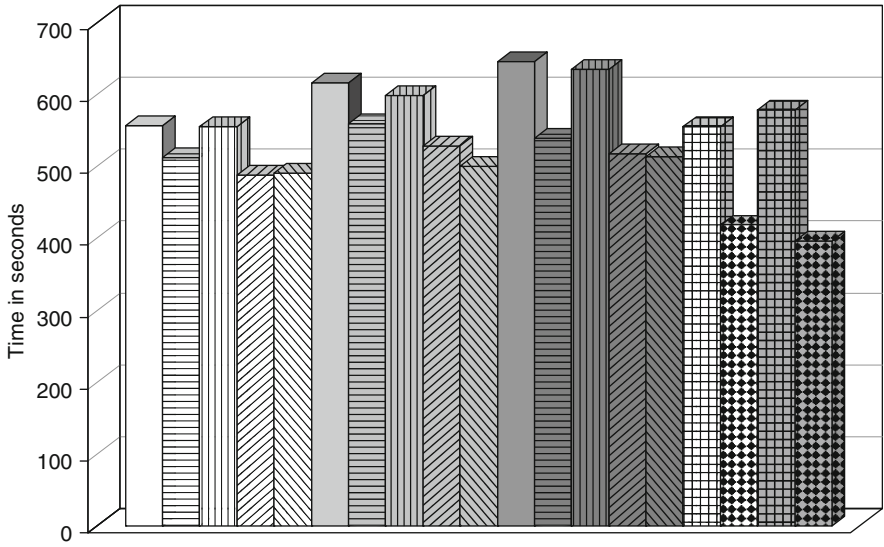


Fig. 3.17 Total times for index construction for one million SP²B triples

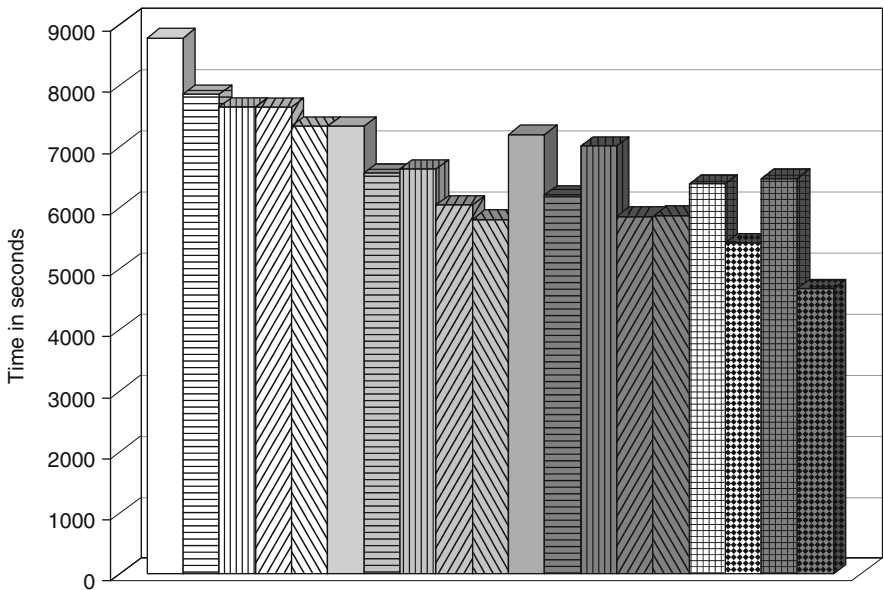


Fig. 3.18 Total times for index construction for ten million SP²B triples

the Barton dataset (MIT 2007). We use the complete Yago dataset, which contains approximately 40 million triples.

We have used another test system for this dataset in order to show that the results remain valid (except of constant factors) even if we use other computer configurations.

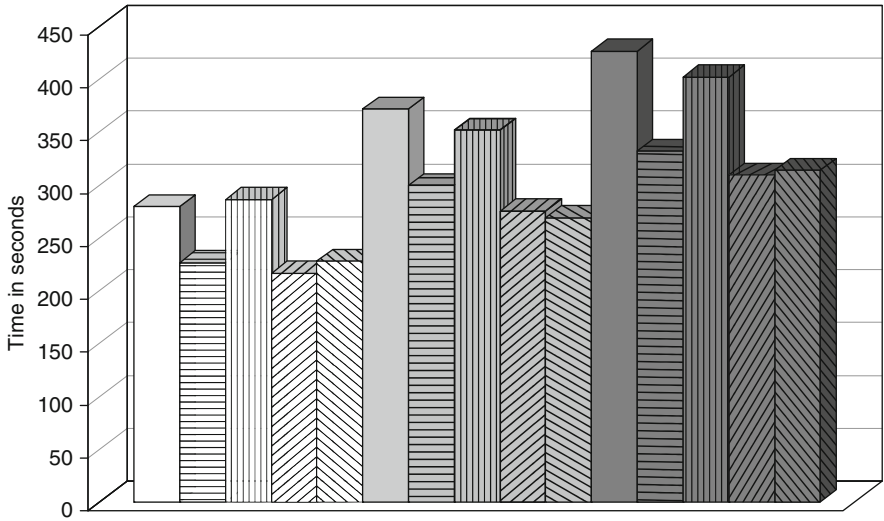


Fig. 3.19 The times for the generation of the initial runs for one million SP²B triples

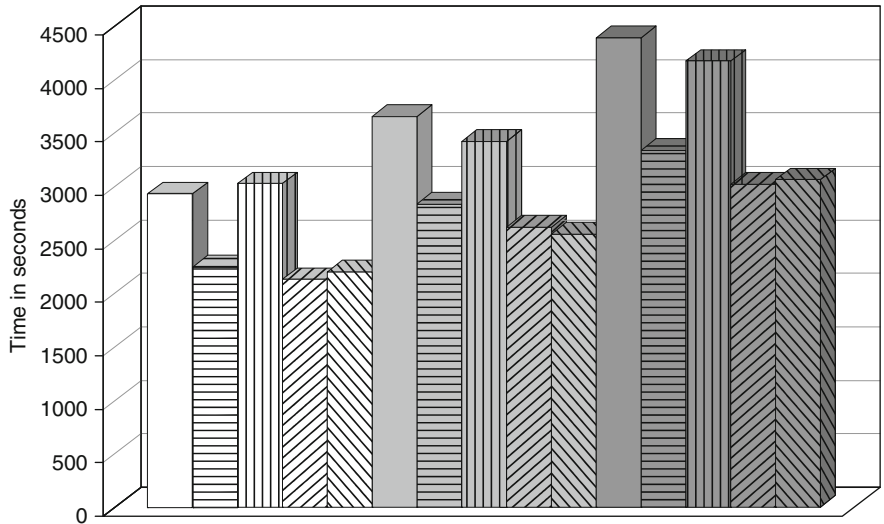


Fig. 3.20 The times for the generation of the initial runs for ten million SP²B triples

The test system for this dataset uses a Dual Quad Core Intel CPU X5550 with 2.67 GHz, 6 GB main memory, Windows XP Professional (x64 Edition) and Java 1.6 64 bit.

We have used the legend presented in Fig. 3.25 for all figures of the Yago dataset. For this large dataset, we tried out Distribution Sort_{Triple Keys} and Distribution

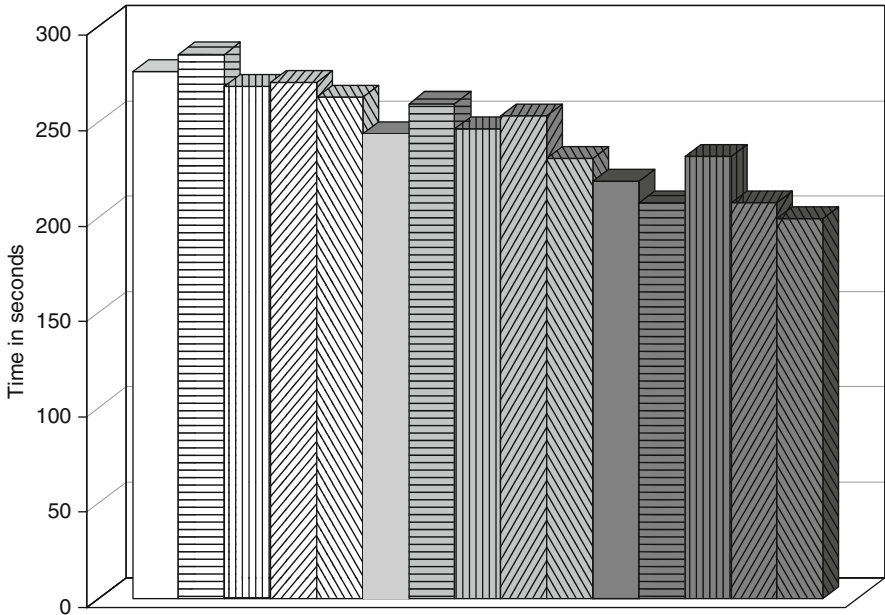


Fig. 3.21 The times for the merging phase for one million SP²B triples

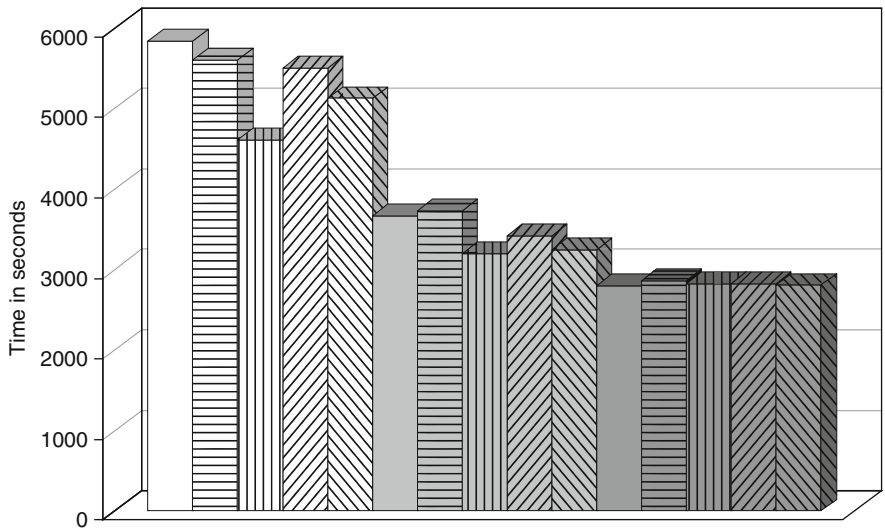


Fig. 3.22 The times for the merging phase for ten million SP²B triples

Sort _{Component Keys} with 1,000 and 10,000 bags. The numbers of distribution passes are 185 for Distribution Sort _{Triple Keys} and 144 for Distribution Sort _{Component Keys} with 1,000 bags, which decrease to 65 for Distribution Sort _{Triple Keys} and 84 for

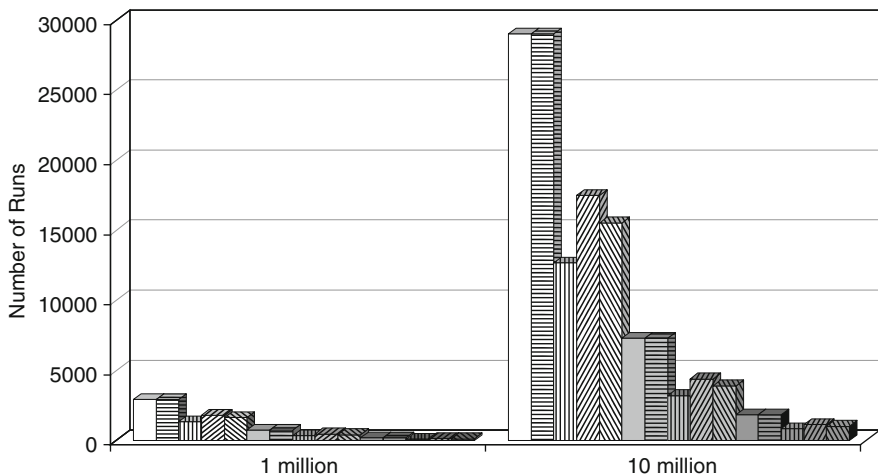


Fig. 3.23 Number of runs using external merge sort approaches for one and ten million SP²B triples

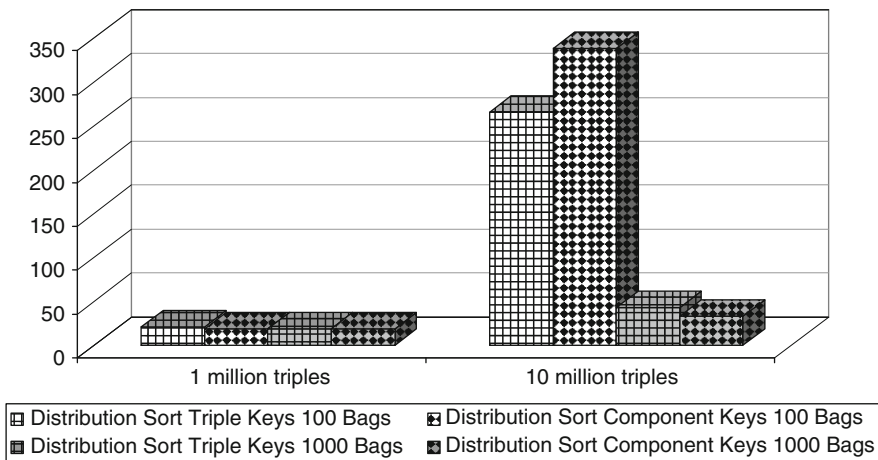


Fig. 3.24 Number of distribution passes using distribution sort approaches for the SP²B datasets

Distribution Sort _{Component Keys} with 10,000 bags. However, each distribution pass takes longer time for 10,000 bags, and thus the total times for 1,000 bags are smaller.

Analogous remarks as for the SP²B dataset apply also to the total times for index construction, the times for the initial runs and for the merging phase, as well as the number of runs (see Figs. 3.26–3.29). Thus, we have verified our experimental results for the synthetic SP²B data using the real-world dataset Yago.

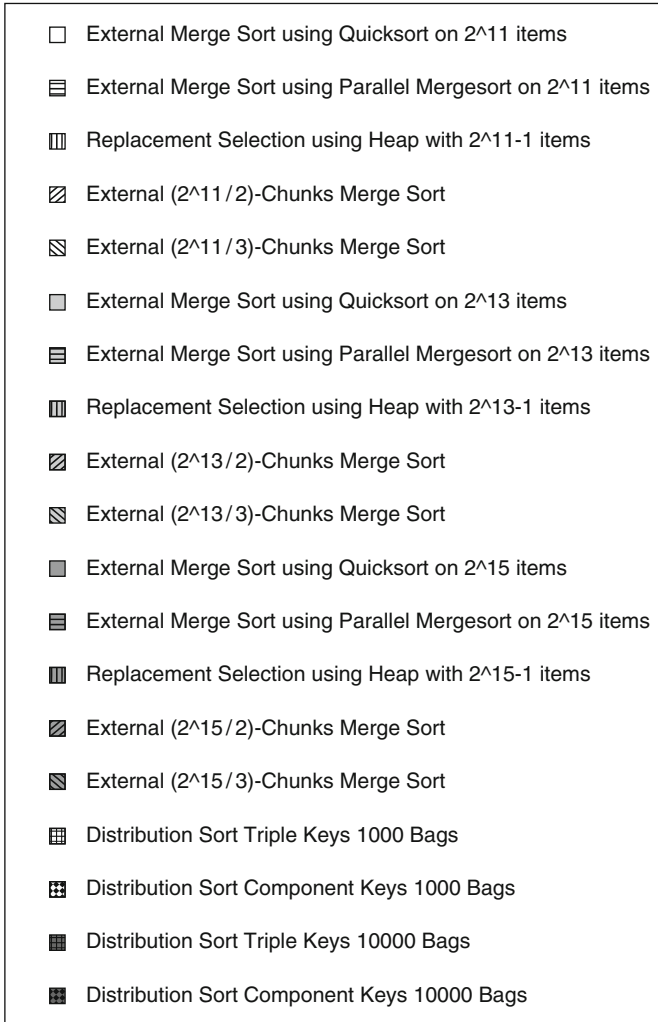


Fig. 3.25 Legend used in the figures of the Yago experiments

3.10 Summary and Conclusions

The state-of-the-art data structures used for indices of large-scale datasets are B^+ -trees. B^+ -trees have a constant main-memory demand for basic operations such as searching for, insertion, and deletion of elements, are optimized for block devices such as hard disks, and are self-balancing having a height logarithmic to the number of inserted key-value pairs. The height of a B^+ -tree is practically a small number often below 5 even for very large datasets.

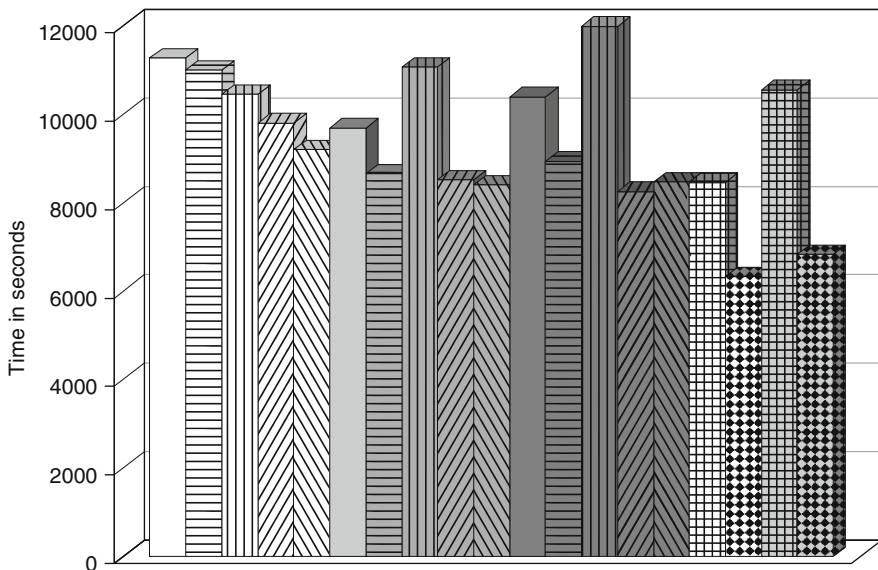


Fig. 3.26 Total times for index construction for the 40-million Yago dataset

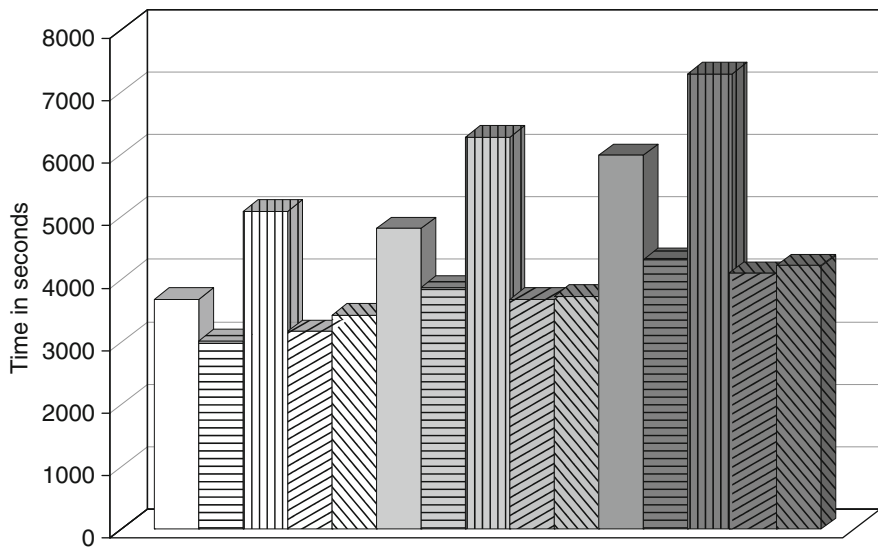


Fig. 3.27 The times for the generation of the initial runs for the 40-million Yago dataset

The performance of index construction for large Semantic Web databases heavily relies on the efficiency of sorting. We propose an efficient variant of the external merge sort algorithm, which stores and retrieves chunks from a special chunks heap in order to speed up replacement selection. We also develop variants of distribution sort specialized for RDF index construction, which greatly speed up

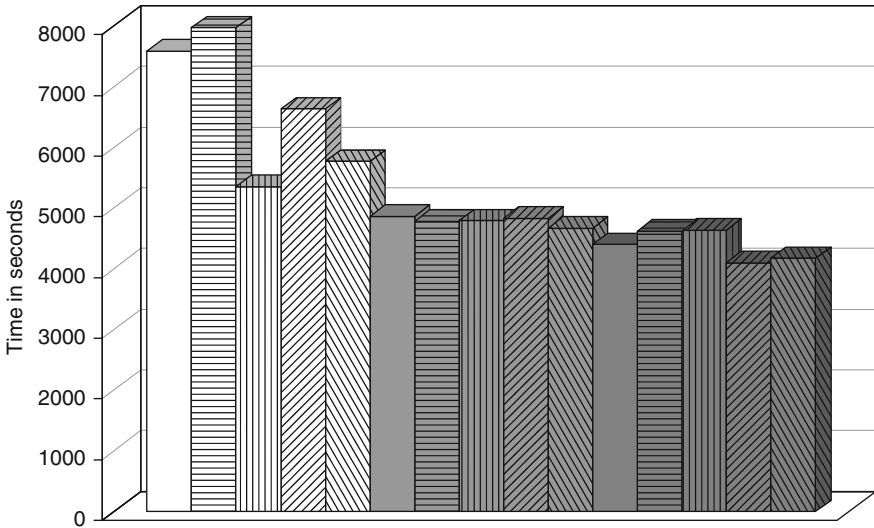


Fig. 3.28 The times for the merging phase for the 40-million Yago dataset

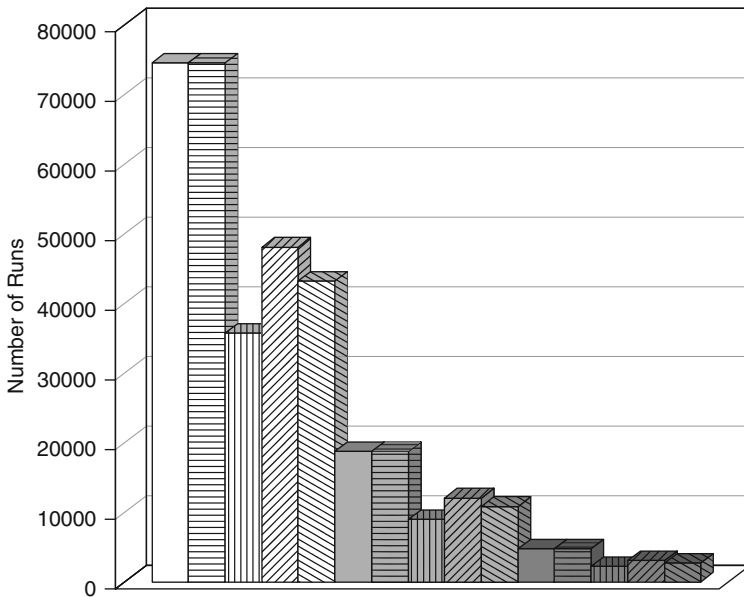


Fig. 3.29 Number of runs using external merge sort approaches for the Yago dataset

RDF sorting by reducing passes through RDF data and by exploiting the RDF-specific properties. These approaches, especially the specialized Distribution Sort Component Key, significantly improve the performance of RDF index construction, as shown by our experimental results.

Chapter 4

Query Processing Overview

Abstract We first present our LUPOSDATE system, including its indexing methods for data management and query engines for query evaluation. Afterward, we describe the different phases of query processing performed by these query engines on a high-level basis. In this chapter, we describe the phase of eliminating redundant language constructs of SPARQL queries in detail. The other (more complex) phases will be described in detail in their own chapters.

4.1 The LUPOSDATE System

In the LUPOSDATE project, we have developed a Semantic Web database system with logically and physically optimized SPARQL engines, named as LUPOSDATE system. Figure 4.1 presents the functionalities of our LUPOSDATE system.

In order to present the functionalities of the LUPOSDATE system, we have developed an online demonstration (Groppe and Groppe 2009; Groppe et al. 2009b), which is available at <http://www.ifis.uni-luebeck.de/index.php?id=luposdate-demo>. The online demonstration visualizes the evaluation of SPARQL queries, and various optimizing techniques used by our system. Although our SPARQL engines work for input data with over one billion triples, due to technical limitations of Java applets, the online demonstration only works for input sizes, which fit into main memory. Figure 4.2 presents a screenshot of the web demonstration. A more detailed description of the web demonstration is available on the demonstration webpage.

In summary, our system and SPARQL engines

- Support full SPARQL 1.0 and run the over 200 W3C test cases (Feigenbaum 2008) successfully,
- Support various approaches to managing RDF data and processing SPARQL queries,

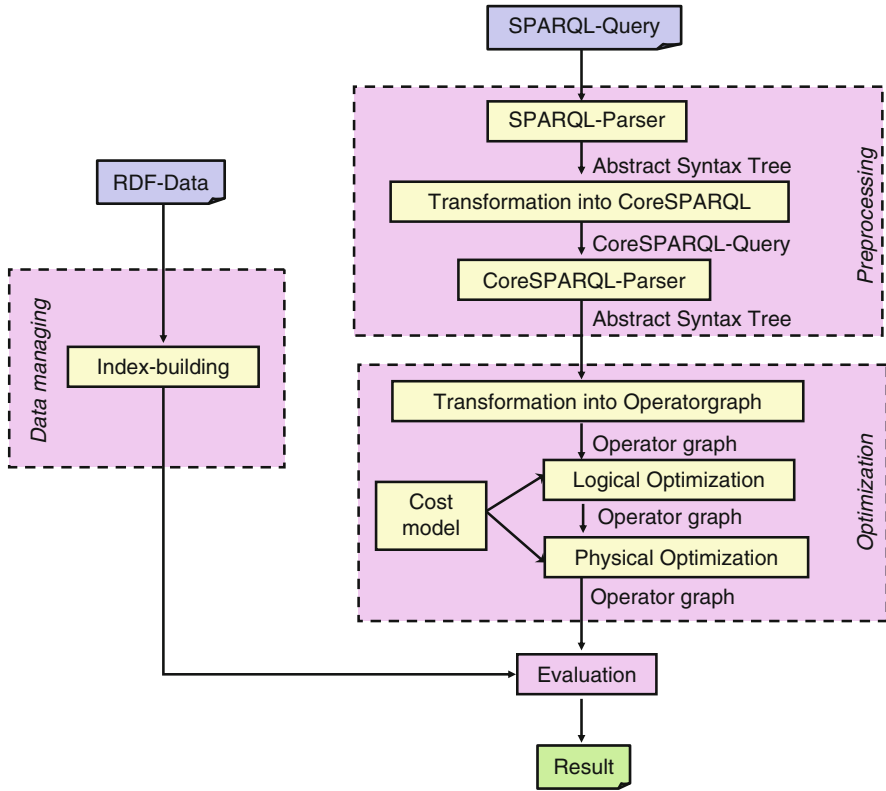


Fig. 4.1 Functionalities and overview of query processing phases of the LUPOSDATE system

- Include more optimizing techniques than existing SPARQL engines:
 - Our own optimization strategies (see Groppe et al. 2009a, b; Groppe et al. 2007a, b),
 - Existing indexing approaches (see Neumann et al. 2008, 2009; Weiss et al. 2008)
- Integrate the existing tools Jena (Wilkinson et al. 2003) and Sesame (Broekstra et al. 2002) for comparison matters,
- Support optimized in-memory (for small data sizes) (Groppe et al. 2009a, 2007a) and disk-based (for large data sizes, for example, over one billion triples) data processing. When processing large-scale datasets, Jena and Sesame engines mainly depend on the existing database techniques.
- Support SPARQL processing of RDF streams (Groppe et al. 2007a), which is the first stream-based engine for SPARQL queries.
- Optimize index construction (Groppe and Groppe 2009).
- Support the visual editing of data and queries.

The prototypes of our SPARQL engines are well tested: we have run them against the test suite of the W3C (Feigenbaum 2008), which contains over 200 test cases, each of which is successfully processed by our prototypes. Our streaming engine

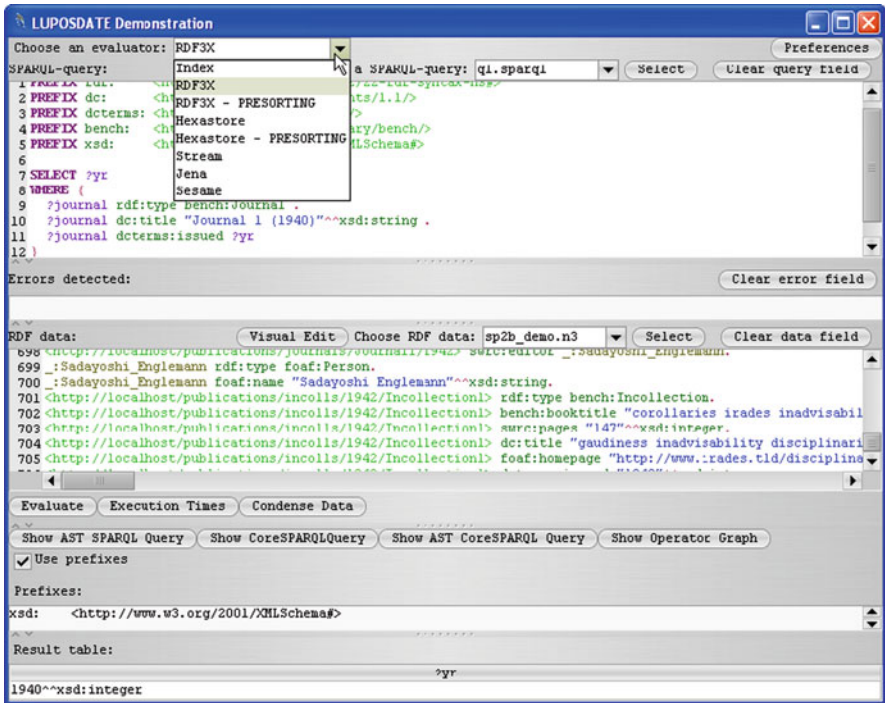


Fig. 4.2 Snapshot of the main window of our online demo

currently does not support named graphs, and thus it runs all W3C test cases successfully except of those with named graphs. Furthermore, we have already successfully used them, for example, in tutorials for master students in our lectures *Mobile und Verteilte Datenbanken (Mobile and Distributed Databases)* (see Grope 2009, 2010).

4.2 Phases of Query Processing

Figure 4.1 presents the phases of query processing in the LUPOSDATE system, which are similar to the ones in other typical database systems such as relational, deductive, and XML databases.

After creating an empty database, RDF data are typically read in and indices are constructed for a faster access to the data for querying and updates. We discuss different types of indices in later chapters. Updates of data are performed on these indices. In comparison, a stream query evaluator does not construct any index of all the data. Stream query evaluators work on a (possibly infinite) stream of data and start to process queries and returning partial results once partial data are available.

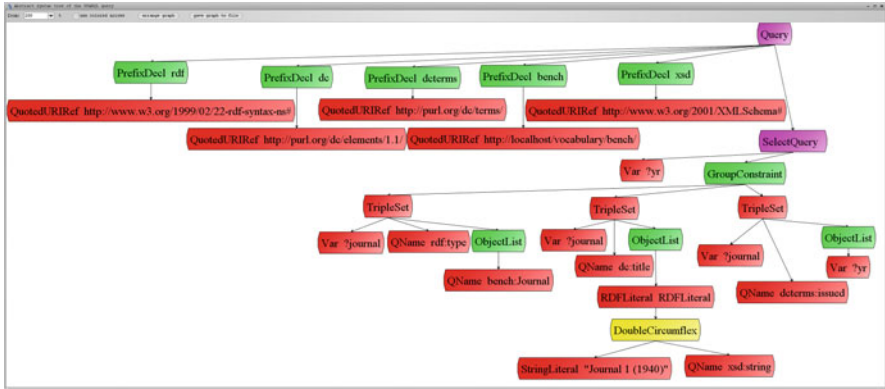


Fig. 4.3 Abstract syntax tree of the SPARQL query in Fig. 4.2

```

CoreSPARQLQuery
SELECT
  ?yr
WHERE
{
  ?journal <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://localhost/vocabulary/bench/Journal> .
  ?journal <http://purl.org/dc/elements/1.1/title> "Journal 1 (1940)"^^<http://www.w3.org/2001/XMLSchema#string> .
  ?journal <http://purl.org/dc/terms/issued> ?yr .
}

```

Fig. 4.4 A CoreSPARQL query transformed from the SPARQL query in Fig. 4.2

In query processing, a query is first parsed according to the grammar of the query language, and an abstract syntax tree is generated and serves as input for the next phase (see Fig. 4.3 for the abstract syntax tree of the SPARQL query in Fig. 4.2 displayed with our LUPOSDATE system). However, sometimes in the case of simple transformations the abstract syntax tree is not explicitly generated (to save processing and space costs) and the output of the next phase is generated just directly during parsing.

Query languages such as SPARQL often support different language constructs with the same semantics, and this complicates later phases since they must consider a large number of different cases. In order to simplify subsequent processing, we define a core of the query language. This core fragment excludes redundant language constructs, but possesses the same expressive power as the original query language. We name this core language of SPARQL the *CoreSPARQL language* (Groppe et al. 2009d) (see Fig. 4.4 for the CoreSPARQL query and Fig. 4.5 for its abstract syntax tree displayed in our LUPOSDATE system). The following subsection describes more details about this CoreSPARQL language and the transformation from any SPARQL query into a CoreSPARQL query.

In the next phase, the query is transformed into an operator graph consisting of logical operators (see Fig. 4.6 for the logical operator graph displayed in our LUPOSDATE system). The semantics of logical operators, that is, their results, is formally defined (see Chap. 5). However, concrete implementations and algorithms, which describe *how* to determine their results, are not defined for logical operators. During the logical optimization phase, the logical operator graph is optimized based on logical

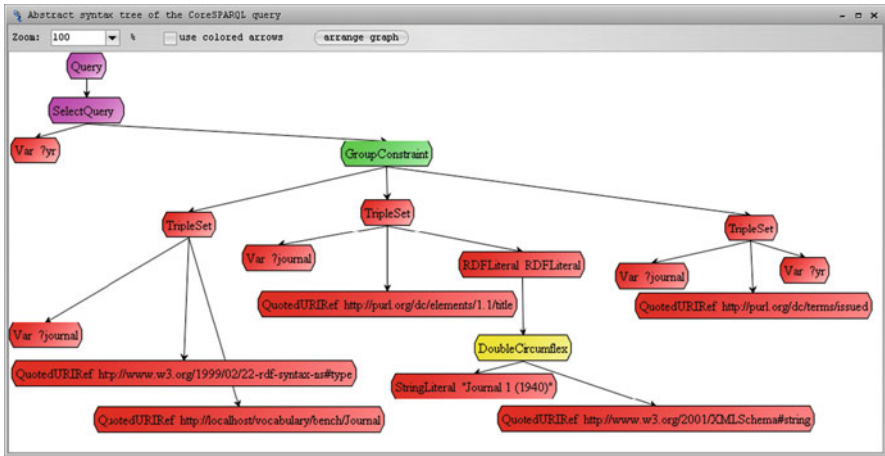


Fig. 4.5 Abstract syntax tree of the CoreSPARQL query of Fig. 4.4

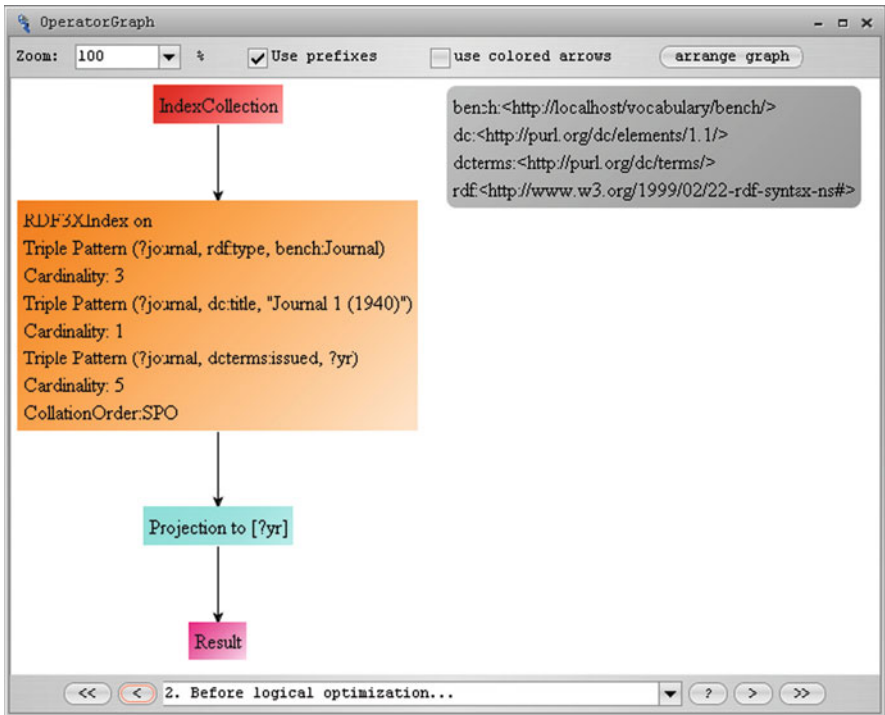


Fig. 4.6 Logical operator graph *before* logical optimization of the SPARQL query in Fig. 4.2

equivalence rules. For example, a filter operator discards solutions based on a Boolean expression and thus usually reduces the number of solutions. If a filter is evaluated more early, then we save processing and space costs: Therefore, a filter operator should

be moved as much before other operations in the logical operator graph as possible. The join order plays another important role in the performance of query processing.

Whereas *quicksort* is known to be one of the fastest (simple) sorting algorithms, *merge sort* can be well parallelized and *insertion sort* is known to be superior for sorting very few data items. Thus, depending on the properties of the data to be sorted and maybe the context of the sort operation like the hardware configuration, a sorting algorithm is sometimes faster or slower in comparison to the other sorting algorithms. Analogously, other logical operators than sorting like those for joins or duplicate elimination can be implemented using many different algorithms, which are slower or faster depending on the characteristics of their input data and their context. Therefore, an important task is to choose concrete implementations (*physical operators*) for each logical operator in the physical optimization phase with the goal to optimize the performance. The choice of physical operators is based on estimations about the data to be processed and the context of the logical operator. The output of this phase is the physical operator graph consisting of physical operators (see Fig. 4.7 for the physical operator graph of the SPARQL query in Fig. 4.2 displayed in our LUPOSDATE system).

Finally, the physical operator graph (also called *execution plan*) is executed and the query result is retrieved (see lower part in Fig. 4.2 for the query result).

Whereas the logical and physical optimizations are the topics of the next two chapters, we describe the first phase transforming any SPARQL query into its equivalent CoreSPARQL query in the next subsection.

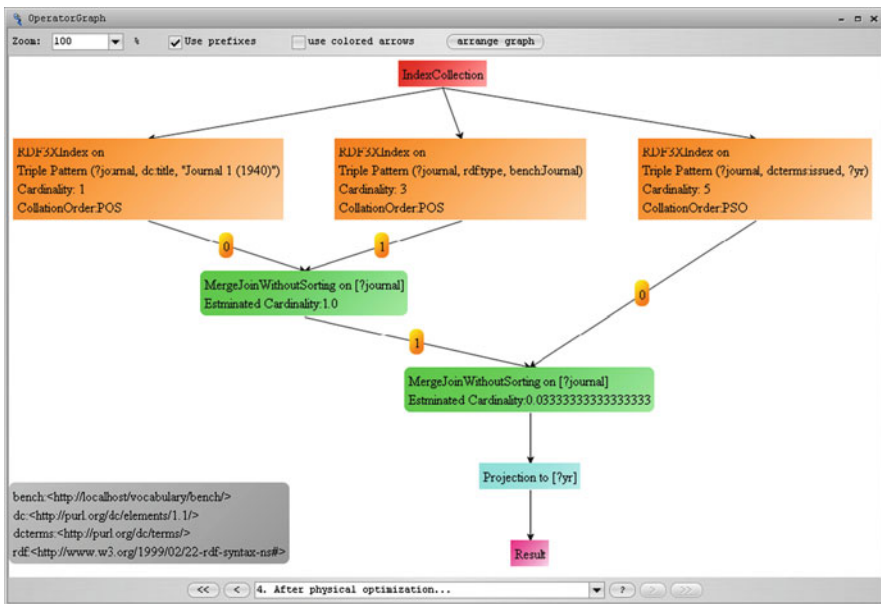


Fig. 4.7 Physical operator graph after physical optimization for the SPARQL query in Fig. 4.2

4.3 CoreSPARQL

SPARQL supports a large number of different language constructs. For example, the three expressions of SPARQL in Fig. 4.8 have the same semantics. Redundant expressive power brings the flexibility of expressiveness and abbreviations bring the simplification of expressions, but they do not increase the expressive power of the language. It is also obvious that the syntax for Expression 1 is user friendly, but Expression 3 is more easily to be interpreted by a machine.

In order to reduce the number of cases, which must be considered when processing SPARQL queries, and in order to make SPARQL queries more machine-processable, we suggest the CoreSPARQL language, which is a core fragment of the SPARQL language. CoreSPARQL possesses the same expressive power as SPARQL, but eliminates redundant language constructs of SPARQL and only allows machine-friendly syntax. We develop an approach, which automatically transforms SPARQL queries into CoreSPARQL queries.

4.3.1 Defining CoreSPARQL

In Definition 1, we describe CoreSPARQL in terms of the common and different properties with SPARQL. A grammar for the syntax of CoreSPARQL is given in a later subsection.

Definition 1 (CoreSPARQL). *CoreSPARQL is a core fragment of SPARQL. A CoreSPARQL query is also a SPARQL query. CoreSPARQL has the same expressive power as SPARQL, but allows only machine-friendly syntax, and excludes most redundant language constructs. Especially, in CoreSPARQL,*

- All triple patterns are only in the form: *s p o*.
- A group graph pattern cannot directly nest another group graph pattern
- Variable names start only with ?
- Blank nodes [] are not allowed

Expression 1	Expression 2	Expression 3
(1 [?x 3]).	[] rdf:first 1; rdf:rest _:b. _:b rdf:first [?x 3]; rdf:rest rdf:nil.	_:b1 rdf:first 1. _:b1 rdf:rest _:b2. _:b2 rdf:first _:b3. _:b3 ?x 3. _:b2 rdf:rest rdf:nil.

Fig. 4.8 Three SPARQL expressions with same semantics

Fig. 4.9 SPARQL and corresponding CoreSPARQL components

component	SPARQL	CoreSPARQL
triple pattern	s1 p1 o1; p2 \$x.	s1 p1 o1. s1 p2 ?x.
blank node []	[p o].	_:b p o.
group graph pattern	{ {s1 p1 o1} s2 p2 o2. }	{ s1 p1 o1. s2 p2 o2. }

- *RDF collections of the form (...) are not allowed*
- *Neither prefixed IRIs nor IRIs, which are relative to a BASE declaration, are allowed*
- *Abbreviations using the keyword a and () are not allowed*

Figure 4.9 demonstrates several SPARQL and corresponding CoreSPARQL components.

4.3.2 Transforming SPARQL Queries into CoreSPARQL Queries

SPARQL provides user-friendly syntax to write RDF queries, and CoreSPARQL queries are easy to process. Therefore, the next task for us is to find a way to automatically transform SPARQL queries into CoreSPARQL queries. We develop a set of transformation rules, such that a SPARQL query can be transformed into a CoreSPARQL query by recursive application of these rules; that is, if the expression of a left-hand side of a rule occurs in a SPARQL query, it is replaced with the right-hand side of the rule.

In the following paragraphs, we assume *rdf* to be an alias for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

We use the following notation to describe these rules: we write *s* (*s*₁, *s*₂,...), *p* (*p*₁, *p*₂, ...), *o* (*o*₁, *o*₂,...) for the subject, predicate, and object of a triple pattern, *os* (*os*₁, *os*₂, ...) for a list of objects, for example, *os* = *o*₁, *o*₂, *o*₃, ..., *o*_{*m*}, where *m* ≥ 1, and *pos* (*pos*₁, *pos*₂, ...) for predicate-object-lists, for example, *pos* = *p*₁ *os*₁; *p*₂ *os*₂; ...; *p*_{*m*} *os*_{*m*}, where *m* ≥ 1. Note that some patterns in the following rules may be not supported by SPARQL. Such patterns are intermediate results of the transformation and will be translated to standard language constructs after the transformation.

- Rule 1: eliminating Object-Lists:
 - 1.1 *s*₁ *p*₁ *o*₁, *os*. => *s*₁ *p*₁ *o*₁. *s*₁ *p*₁ *os*.

- Rule 2: eliminating Predicate–Object-Lists:
 - 2.1 $s_1 p_1 os_1; pos. \Rightarrow s_1 p_1 os_1. s_1 pos.$

- Rule 3: eliminating blank nodes [].
 - 3.1 $[] \Rightarrow _ :b,$
where b is a blank node label not used elsewhere in the query
 - 3.2 $[pos] \Rightarrow _ :b pos.$
 - 3.3 $[pos_1] pos_2. \Rightarrow _ :b pos_1. _ :b pos_2.$
 - 3.4 $s_1 p_1 [pos]. \Rightarrow s_1 p_1 _ :b. _ :b pos.$
 - 3.5 $s_1 p_1 [pos] p_2 os_1. \Rightarrow s_1 p_1 _ :b. _ :b pos. _ :b p_2 os_1.$

- Rule 4: eliminating RDF collections (), where e (e_1, e_2, \dots) is an element of the collection, that is, a variable, a literal, a blank node, or a collection. Here, we introduce a variant of the collection, for example, $(e)_{s=_ :b}$, to restrict that the blank node, which is allocated for the collection (e), must be $_ :b$. A new blank node $_ :b$ on the right side of a rule must be chosen in such a way that $_ :b$ is not used elsewhere in the query.
 - 4.1 $(e) pos. \Rightarrow _ :b rdf:first e. _ :b rdf:rest rdf:nil. _ :b pos.$
 - 4.2 $(e). \Rightarrow _ :b rdf:first e. _ :b rdf:rest rdf:nil.$
 - 4.3 $(e_1 e_2 e_3 \dots). \Rightarrow _ :b rdf:first e_1. _ :b rdf:rest (e_2 e_3 \dots).$
 - 4.4 $s p (e_1 e_2 \dots). \Rightarrow s p _ :b. (e_1 e_2 \dots)_{s=_ :b}.$
 - 4.5 $s p (e_1 e_2 \dots) pos. \Rightarrow s p _ :b. (e_1 e_2 \dots)_{s=_ :b}. _ :b pos.$
 - 4.6 $(e_1 e_2 \dots)_{s=_ :b}. \Rightarrow _ :b rdf:first e_1. _ :b rdf:rest (e_2 \dots).$
 - 4.7 $(e)_{s=_ :b}. \Rightarrow _ :b rdf:first e. _ :b rdf:rest rdf:nil.$
 - 4.8 $() \Rightarrow rdf:nil$

- Rule 5: eliminate the keyword `a`:
 - 5.1 $a \Rightarrow rdf:type$

- Rule 6: eliminate directly nested group graph patterns
 - 6.1 $\{ \{A\} \dots \} \Rightarrow \{ A \dots \},$
where $\{A\}$ is not a part of an *OPTIONAL*, or an *UNION*, or a *GRAPH* operand; A does not consist of only *Filter* expressions either.
 - 6.2 $\{ \{Filter(e).\} \dots \}$
 $\Rightarrow \begin{cases} \{ Filter(true) \dots \}, & \text{if the result of the static analysis of } e \text{ is true.} \\ \{ Filter(false) \dots \}, & \text{if the result of the static analysis of } e \text{ is false or a type error.} \end{cases}$

For example, the expression $10 > 1$ is statically analyzed to true, and thus $\{Filter(10 > 1)\} = Filter(true)$.

In the group graph pattern $\{Filter(bound(?x))\}$, the variable x will never be bound. Therefore, the static analysis of $bound(?x)$ can produce a type error, and thus $\{Filter(bound(?x))\} = Filter(false)$. For the details on the static analysis and type errors, see Sect. 11.2 Filter Evaluation in the SPARQL specification (Prud'hommeaux and Seaborne 2008).

- Rule 7: eliminating prefixes and BASE declarations.
 - 7.1 $p:a \Rightarrow \langle prefix(p) a \rangle,$

where $\text{prefix}(p)$ is a function to resolve the prefixed IRI $p:a$ according to defined PREFIX and BASE declarations. The PREFIX and BASE declarations are deleted in the CoreSPARQL query.

Example 1 Using this example, we demonstrate how to transform a SPARQL expression $t1 = (1 [p \ o1] (2))$. into the corresponding CoreSPARQL expression by recursively applying the rules above.

1. Applying Rule 4.3 on $t1$: $t1 \Rightarrow t2, t3$.:
 - $_:b1 \text{ rdf:first } 1.$ (t2)
 - $_:b1 \text{ rdf:rest } ([p \ o1] (2)).$ (t3)
2. Applying Rule 4.4 on $t3$: $t3 \Rightarrow t4, t5$.:
 - $_:b1 \text{ rdf:rest } _:b2.$ (t4)
 - $([p \ o1] (2))_s=_:b2.$ (t5)
3. Applying Rule 4.6 on $t5$: $t5 \Rightarrow t6, t7$.:
 - $_:b2 \text{ rdf:first } [p \ o1].$ (t6)
 - $_:b2 \text{ rdf:rest } ((2)).$ (t7)
4. Applying Rule 3.4 on $t6$: $t6 \Rightarrow t8, t9$.:
 - $_:b2 \text{ rdf:first } _:b3.$ (t8)
 - $_:b3 \text{ po } 1.$ (t9)
5. Applying Rule 4.4 on $t7$: $t7 \Rightarrow t10, t11$.:
 - $_:b2 \text{ rdf:rest } _:b4.$ (t10)
 - $((2))_s=_:b4.$ (t11)
6. Applying Rule 4.7 on $t11$: $t11 \Rightarrow t12, t13$.:
 - $_:b4 \text{ rdf:first } (2).$ (t12)
 - $_:b4 \text{ rdf:rest } \text{rdf:nil}.$ (t13)
7. Applying Rule 4.4 on $t12$: $t12 \Rightarrow t14, t15$.:
 - $_:b4 \text{ rdf:first } _:b5.$ (t14)
 - $(2)_s=_:b5.$ (t15)
8. Applying Rule 4.7 on $t15$: $t15 \Rightarrow t16, t17$.:
 - $_:b5 \text{ rdf:first } 2.$ (t16)
 - $_:b5 \text{ rdf:rest } \text{rdf:nil}.$ (t17)

The transformation result consists of triple patterns $t2, t4, t8, t9, t10, t13, t14, t16,$ and $t17$, where additionally the prefixed form of IRIs have been replaced with their long form according to Rule 7.1.

Note that there are further redundancies, which we allow in CoreSPARQL, as they can be processed in a machine-friendly way. Nevertheless, we explain these redundancies and how to eliminate these redundancies in the following paragraphs.

The wildcard $*$ can be replaced by the concrete list of variables in SELECT [DISTINCT | REDUCED] $*$ and DESCRIBE $*$; that is, the SPARQL expression can be replaced by SELECT [DISTINCT | REDUCED] $\text{Var}_1, \dots, \text{Var}_n$ and DESCRIBE $\text{Var}_1, \dots, \text{Var}_n$, where $\text{Var}_1, \dots, \text{Var}_n$ are all variables of the original SPARQL query, which have been bound in triple patterns.

Furthermore, the REDUCED keyword can be replaced by DISTINCT or can be deleted, as SELECT REDUCED ... allows any number of duplicates between the number of duplicates of SELECT ... and SELECT DISTINCT ...

Any operations on constants can be replaced by the result of their applications.

According to (Gutierrez et al. 2004), blank node labels of the form `_:b` can be replaced by a local variable `?_b`, where the variable `?_b` must not be used in the original SPARQL query. Whenever at least one blank node label is replaced, SELECT [DISTINCT | REDUCED] * and DESCRIBE * must be replaced with SELECT [DISTINCT | REDUCED] Var₁, ..., Var_n and DESCRIBE Var₁, ..., Var_n, where Var₁, ..., Var_n are all variables of the original SPARQL query, which have been bound in triple patterns; that is, all local variables `?_b` do not occur in the final result. Note that unbound variables in FILTER expressions raise type errors.

4.3.3 CoreSPARQL Grammar

The grammar rules for CoreSPARQL are adapted from the grammar rules for SPARQL, which are given in A.8 Grammar in Prud'hommeaus and Seaborne (2008). Twenty-two rules (2–4, 32–42, 48, 68, 71–72, 75, 92, and 99–100) in SPARQL are not needed and eliminated; ten rules (1, 21, 22, 25, 31, 37, 44, 45, 67, and 69) differ from the corresponding ones in SPARQL; and the rest remains unchanged. Here, we only present these ten adapted rules in Table 4.1 and comment

Table 4.1 Adapted grammar rules of the SPARQL grammar

[1]	<i>Query</i>	::=	SelectQuery ConstructQuery DescribeQuery AskQuery
PREFIX and BASE declarations are not allowed.			
[21]	<i>TriplesBlock</i>	::=	VarOrTerm VarOrIRIref VarOrTerm
Only allow triple patterns of the form s p o.			
[22]	<i>GraphPatternNotTriples</i>	::=	OptionalGraphPattern UnionGraphPattern GraphGraphPattern
Group graph patterns are not allowed to directly nest any other group graph pattern.			
[25]	<i>UnionGraphPattern</i>	::=	GroupGraphPattern 'UNION' GroupGraphPattern
Group graph patterns are not allowed to directly nest any other group graph pattern.			
[31]	<i>ConstructTriples</i>	::=	TriplesBlock ('?' ConstructTriples?)?
Only allow triple patterns of the form s p o.			
[37]	<i>Verb</i>	::=	VarOrIRIref
The keyword a is not allowed anymore.			
[44]	<i>Var</i>	::=	VAR1
Variable names start only with “?”.			
[45]	<i>GraphTerm</i>	::=	IRIref RDFLiteral NumericLiteral BooleanLiteral BlankNode
The abbreviation () for rdf:nil is not allowed.			
[67]	<i>IRIref</i>	::=	IRI_REF
Prefixed IRIs are not allowed anymore.			
[69]	<i>BlankNode</i>	::=	BLANK_NODE_LABEL
Blank nodes are represented only with label.			

the consequence of the adaptation. We use the same number and the name of left-hand side for every rule as in SPARQL, except for Rule 25. The name of left-hand side of Rule 25 is changed from *GroupOrUnionGraphPattern* to *UnionGraphPattern* in order to reflect the prohibition of the direct nesting of group graph patterns.

4.4 Related Work

Jarke and Koch (1984), Ioannidis (1996), and Chaudhuri et al. (1998) give an overview of logical transformation techniques and that of physical evaluation methods for database queries using the framework of the relational algebra (Ioannidis 1996; Chaudhuri et al. 1998) or of the (tuple) relational calculus (Jarke and Koch 1984). In these works, the relational query (e.g., an SQL query) is first transformed into a relational algebra tree (Ioannidis 1996; Chaudhuri et al. 1998) or into an object graph (Jarke and Koch 1984), on which logical transformation rules are applied in order to optimize the query evaluation. After that, depending on cost estimations, physical operators are chosen for the logical operators in the relational algebra tree (Ioannidis 1996; Chaudhuri et al. 1998), or in the object graph (Jarke and Koch 1984), in order to finally evaluate the query in the estimated fastest way. The physical operators are chosen from a set of different implementations with different runtime performance. These techniques, which especially use the framework of the relational algebra, are also presented in various standard works for database systems (e.g., Garcia-Molina et al. 2002; Connolly and Begg 2002; Elmasri and Navathe 2000).

We propose the CoreSPARQL fragment in (Groppe et al. 2009d).

4.5 Summary and Conclusions

We suggest the CoreSPARQL language, which is a core fragment of SPARQL, but has the same expressiveness as SPARQL 1.0. Optimization approaches, SPARQL engines, and all applications, which process SPARQL queries, benefit from CoreSPARQL: CoreSPARQL has machine-friendly syntax and thus is easy to process; CoreSPARQL contains less language constructs and thus reduces the number of cases to be considered.

We apply the logical and physical optimization framework in our SPARQL engines. Once the logical operator graph has been generated, logical equivalence rules can be used to optimize the performance. After logical optimization, the physical optimization chooses the best estimated physical operator for each logical operator for processing joins, sort operations, optional expressions, and index scans to determine the result of a triple pattern. The physical operator graph, also called execution plan, is finally executed to retrieve the query result.

Chapter 5

Logical Optimization

Abstract In this chapter, we first introduce an algebra for SPARQL queries and define the semantics of query evaluation. Afterward, we present equivalency rules for optimizing the query processing and present a heuristic approach to query optimization based on these equivalency rules. Afterward, we deal with query optimizers, which enumerate all possible query plans and choose the one with the best estimated costs. Finally, we describe how to employ histograms for estimating the cardinality of operator results as basis for cost estimations.

5.1 Logical Algebra

In order to define the semantics of query evaluation, we formalize an algebra of the core fragment of SPARQL over simple RDF, that is, RDF without RDFS vocabulary and literal rules. We extend the algebraic formalization of Pérez et al. (2006) by considering a larger fragment of SPARQL and by allowing more complex built-in conditions and empty graph patterns. With our extensions, the algebra covers the full power of SPARQL 1.0 (Prud'hommeaux and Seaborne 2008), such that even the different types of SPARQL queries like *SELECT*, *ASK*, and *CONSTRUCT* queries can be expressed in this algebra.

A SPARQL query is evaluated on RDF graphs. In order to describe the semantics of SPARQL queries, we first define built-in conditions of SPARQL, which are used in *FILTER* clauses of SPARQL. Note that we do not distinguish between variables and blank node variables any more, as blank node variables can be eliminated as described in the transformation from SPARQL queries into Core-SPARQL queries.

Definition 1 (Built-in condition). *We assume the existence of a set V of variables, which is disjoint from the set of IRIs I , the set of blank nodes B , and the set of literals L . A built-in condition is recursively defined as follows:*

- (a) $l \in L$, $i \in I$ and $v \in V$ are built-in conditions
- (b) If R_1 and R_2 are built-in conditions, then

- I. The Boolean expressions $R_1 \wedge R_2$, $R_1 \vee R_2$, $\neg R_1$ and R_1 op R_2 , where op $\in \{=, \neq, <, \leq, >, \geq\}$, are also built-in conditions,
 - II. The numeric expressions R_1 op R_2 , where op $\in \{+, -, *, / \}$, are also built-in conditions, and
- (c) If R_1, \dots, R_n are built-in conditions, then $\text{func}(R_1, \dots, R_n)$ is also a built-in condition, where func is an external or a SPARQL built-in function. See Prud'hommeaux and Seaborne (2008) for a complete list of SPARQL built-in functions.

The core component of SPARQL queries is a graph pattern.

Definition 2 (Graph pattern). A graph pattern is defined recursively as follows:

- (a) The empty graph pattern $\{ \}$ is a graph pattern.
- (b) A triple pattern tp , where $\text{tp} \in (I \cup B \cup L \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$, is a graph pattern
- (c) If P_1 and P_2 are graph patterns, then $P_1 \text{ AND } P_2$, $P_1 \text{ OPT } P_2$, and $P_1 \text{ UNION } P_2$ are graph patterns
- (d) If P is a graph pattern and $v_i \in I \cup V$, then $\text{GRAPH } v_i (P)$ is also a graph pattern
- (e) If P is a graph pattern and R is a built-in condition, then $P \text{ FILTER } R$ is a graph pattern

The list of graph patterns has been extended with $\text{GRAPH } v_i (P)$ in comparison to Pérez et al. (2006) for the support of named graphs. Note that the SPARQL syntax (Prud'hommeaux and Seaborne 2008) differs slightly from the above notation. For example, the SPARQL expression $P_1 \text{ P}_2$ in Prud'hommeaux and Seaborne (2008) corresponds to our graph pattern $P_1 \text{ AND } P_2$, $P_1 \text{ OPTIONAL } P_2$ to $P_1 \text{ OPT } P_2$, $\text{GRAPH } v_i \{P\}$ to $\text{GRAPH } v_i (P)$, and P . and $\{P\}$ to P . Furthermore, the SPARQL operators $\&\&$, \parallel , $!$, $!=$, $<=$ and $>=$ correspond to the operators \wedge , \vee , \neg , \neq , \leq and \geq in the built-in conditions (see Definition 1). SPARQL also supports some further redundant equivalent language constructs like object lists and predicate-object lists in triple patterns, and several other abbreviations, which can be transformed into an equivalent long form (Groppe et al. 2009d).

Note that SPARQL allows literals as subjects of triple patterns, but RDF does not allow literals as subjects of triples. Therefore, the triple patterns with literal subjects always return the empty result for any input RDF graph, and thus this kind of triple patterns is unsatisfiable.

In the following paragraphs, we further extend the algebraic formalization of Pérez et al. (2006) for considering also the different types of SPARQL queries, that is, SELECT, ASK, and CONSTRUCT queries, and their modifiers such as DISTINCT, ORDER BY, LIMIT, and OFFSET. DESCRIBE queries are not considered here, as their result has only informally been described in the SPARQL specification (Prud'hommeaux and Seaborne 2008) and is implementation-dependent.

Definition 3 (Query heads and extended list of graph patterns). *The following query heads and additional graph patterns are defined as follows:*

- (a) *If P is a graph pattern and $\{v_1, \dots, v_n\}$ is a set of variables, then the projection $\prod_{\{v_1, \dots, v_n\}} P$ is a graph pattern.*
- (b) *If P is a graph pattern, then $\text{DISTINCT } P$ is a graph pattern for the elimination of duplicates.*
- (c) *If R_1, \dots, R_n are built-in conditions, then $\text{SORT}_{\text{ORDER}_1(R_1), \dots, \text{ORDER}_n(R_n)} P$, where $\text{ORDER}_1, \dots, \text{ORDER}_n \in \{\text{ASC}, \text{DESC}\}$ and which corresponds to the ORDER BY clause of SPARQL queries, is a graph pattern.*
- (d) *If P is a graph pattern and i an integer, then $\text{LIMIT } i P$ and $\text{OFFSET } i P$ are graph patterns for the LIMIT and OFFSET clause respectively of SPARQL queries.*
- (e) *If P is a graph pattern, then $\text{ASK } P$ is a query head for ASK queries.*
- (f) *If P is a graph pattern and $\{tp_1, \dots, tp_n\}$ a set of triple patterns tp_1, \dots, tp_n , then $\text{CONSTRUCT}\{tp_1, \dots, tp_n\} P$ is a query head for CONSTRUCT queries.*

Example 1 (Graph pattern and SPARQL query). The graph pattern

$$\text{DISTINCT } \prod_{\{?title, ?price\}} ((?x \text{ dc:title } ?title) \text{ OPT } (?x \text{ ns:price } ?price) \text{ FILTER}(?price < 30)))$$

represents the SPARQL query

```
SELECT DISTINCT ?title ?price
WHERE {
    ?x dc:title ?title.
    OPTIONAL { ?x ns:price ?price.
    FILTER (?price < 30) }.
```

The transformation from any CoreSPARQL query into its graph pattern or its query head respectively is straightforward and is thus left to the reader.

5.1.1 Semantics of the Logical Algebra Operators

In this section, we define the semantics of SPARQL queries by describing the result of graph patterns (see Definition 2) evaluated on an RDF graph. For this purpose, we need to introduce several concepts.

Definition 4 (Binding). *A binding is a tuple (v, t) , where $v \in V$ represents a SPARQL variable and $t \in (I \cup B \cup L)$ one of its values.*

A solution contains one element of the result of a SPARQL graph pattern.

Definition 5 (Solution). *A solution E is a set of bindings, where each variable in the bindings has exactly one assigned value; that is, $\forall (v, t) \in E: v \in V \wedge t \in (I \cup B \cup L)$ and $\forall (v_1, t_1) \in E: \forall (v_2, t_2) \in E: (v_1 \neq v_2 \vee t_1 = t_2)$.*

With this, we can define the result of a built-in condition.

Definition 6 (Result of built-in condition). *The result $\text{Eval}_R(E)$ of a built-in condition R evaluated on a solution E is a Boolean value, that is, true or false, and is determined recursively according to the following rules:*

- (a) $R = R_1 \text{ op } R_2$, where $\text{op} \in \{\wedge, \vee\}$: $\text{Eval}_{R_1 \text{ op } R_2}(E) = \text{Eval}_{R_1}(E) \text{ op } \text{Eval}_{R_2}(E)$
- (b) $R = R_1 \text{ op } R_2$, where $\text{op} \in \{=, \neq, <, \leq, >, \geq\}$: $\text{Eval}_{R_1 \text{ op } R_2}(E) = \text{Value}_{R_1}(E) \text{ op } \text{Value}_{R_2}(E)$, where $\text{Value}_R(E)$ is defined recursively as follows:
 - I. $R \in L$ or $R \in I$: $\text{Value}_R(E) = R$
 - II. $R \in V$: $\text{Value}_R(E) = t$ if $(R, t) \in E$ or $\text{Value}_R(E)$ raises an error if $\nexists t^*$: $(R, t^*) \in E$ [see Sect. 11.2 of Prud'hommeaux and Seaborne (2008) for handling errors]
 - III. $R = \text{op } R_1$, where $\text{op} \in \{+, -, \neg\}$: $\text{Value}_R(E) = \text{op } \text{Value}_{R_1}(E)$
 - IV. $R = R_1 \text{ op } R_2$, where $\text{op} \in \{+, -, *, /, \wedge, \vee, =, \neq, <, \leq, >, \geq\}$: $\text{Value}_{R_1 \text{ op } R_2}(E) = \text{Value}_{R_1}(E) \text{ op } \text{Value}_{R_2}(E)$
 - V. $R = \text{func}(R_1, \dots, R_n)$: $\text{Value}_{\text{func}(R_1, \dots, R_n)}(E) = \text{func}(R_1, \dots, R_n)$, where $\text{func}(R_1, \dots, R_n)$ is an external or a SPARQL built-in function. For example, $\text{bound}(v)$ returns true if $\exists t:(v, t) \in E$; otherwise, $\text{bound}(v)$ returns false. See Prud'hommeaux and Seaborne (2008) for a complete list of SPARQL built-in functions.
- (c) Otherwise: $\text{Eval}_R(E) = \text{xsd:boolean}(\text{Value}_R(E))$ [see Prud'hommeaux and Seaborne (2008) for the Effective Boolean value xsd:boolean].

SPARQL queries can be SELECT, DESCRIBE, ASK, or CONSTRUCT queries. ASK queries return Boolean results. CONSTRUCT queries and DESCRIBE queries return RDF graphs. SELECT queries usually return bags (also called multisets) of solutions. A bag contains unordered entries. Whereas in a set each of its unordered entries can occur only once, each entry in a bag can also occur several times. If a SELECT query contains the DISTINCT modifier, then actually a set of solutions is returned, as duplicates are eliminated. If the SELECT query contains an ORDER BY clause for sorting its result, then a sequence of solutions is returned if the SELECT query contains no DISTINCT modifier, and an ordered set of solutions if the SELECT query contains a DISTINCT modifier.

We denote bags by enclosing its entries with \langle and \rangle . For example, $\langle x_1, x_2 \rangle$ is a bag containing the entries x_1 and x_2 , where the entries x_1 and x_2 are typically solutions in our definitions for graph pattern results. The concatenation of bags is expressed with the operator o , for example, $\langle x_1, x_2 \rangle o \langle x_3 \rangle = \langle x_1, x_2, x_3 \rangle$. The bag X contains the bag Y , denoted by $Y \subseteq X$, if $\forall y \in Y: (y \in Y) \leq (y \in X)$, where $y \in Y$ represents the number of copies of y in Y . Furthermore, the bag $X = \langle x_1, \dots, x_n \rangle$ set-contains the bag $Y = \langle y_1, \dots, y_m \rangle$, denoted by $Y \subseteq_{\text{set}} X$, if $\forall y \in \{y_1, \dots, y_m\}: \exists x \in \{x_1, \dots, x_n\}: x = y$. The difference between two bags X and Y is defined to be $Z: = X - Y \subseteq X$, where $\forall x \in X: x \in Z = \max((x \in X) - (x \in Y), 0)$. We define the set-difference between two bags X and Y , denoted by $Z_{\text{set}}: = X -_{\text{set}} Y$, to be $Z_{\text{set}} = \langle x / x \in \{x_1, \dots, x_n\} \wedge x \notin \{y_1, \dots, y_m\} \rangle$, where the set-difference contains the entries of $X = \langle x_1, \dots, x_n \rangle$ excluding the entries of the bag $Y = \langle y_1, \dots, y_m \rangle$.

In the definitions for the result of graph patterns, we use the operators *Join*, *Union*, and *Left outer-join*, which work on bags of solutions.

Definition 7 (Join, Union and Left outer-join of bags of solutions). Let Ω_1 and Ω_2 be two bags of solutions. Then:

- (a) *Join*: $\Omega_1 \bowtie \Omega_2 = \langle \omega_1 \cup \omega_2 \mid \omega_1 \in \Omega_1 \wedge \omega_2 \in \Omega_2 \wedge \forall (v, t_1) \in \omega_1 \cup \omega_2: \forall (v, t_2) \in \omega_1 \cup \omega_2: t_1=t_2 \rangle$
- (b) *Union*: $\Omega_1 \cup \Omega_2 = \langle \omega \mid \omega \in \Omega_1 \vee \omega \in \Omega_2 \rangle$
- (c) *Left outer-join*: $\Omega_1 \bowtie \Omega_2 = \Omega_1 \bowtie \Omega_2 \circ \langle \omega_1 \mid \omega_1 \in \Omega_1 \wedge \omega_1 \notin \{ \omega_1 \mid \omega_1 \in \Omega_1 \wedge \omega_2 \in \Omega_2 \wedge \forall (v, t_1) \in \omega_1 \cup \omega_2: \forall (v, t_2) \in \omega_1 \cup \omega_2: t_1=t_2 \} \rangle$

Definition 8 (Result of graph patterns). Let D be an RDF graph and $NG = \{D(IRI1), \dots, D(IRIn)\}$ be a set of named graphs, where $IRI1, \dots, IRIn$ are the IRI labels of the named RDF graphs, $(e1 \ e2 \ e3)$ a triple pattern, $P1$ and $P2$ graph patterns, R a built-in condition, IRI an IRI label, and $v \in V$ a variable. Then the evaluation of a graph pattern P over the default RDF graph D and the named RDF graphs NG denoted by $[[P]]_{D, NG}$ is defined recursively as follows:

- a) $[[\{\}]]_{D, NG} = \langle \rangle$
- b) $[[(e1 \ e2 \ e3)]]_{D, NG} = \langle E \mid (d_1, d_2, d_3) \in D \wedge E = \{ (x, v) \mid i \in \{1, 2, 3\} \wedge x = e_i \wedge e_i \in V \wedge v = d_i \} \wedge ((\forall j \in \{1, 2, 3\}: (e_j \in V) \vee (e_j = d_j)) \wedge \forall (n, v_1) \in E: \forall (n, v_2) \in E: v_1 = v_2) \rangle$
- c) $[[P1 \text{ AND } P2]]_{D, NG} = [[P1]]_{D, NG} \bowtie [[P2]]_{D, NG}$
- d) $[[P1 \text{ OPT } P2]]_{D, NG} = [[P1]]_{D, NG} \bowtie ([[P1]]_{D, NG} \bowtie [[P2]]_{D, NG})$
- e) $[[P1 \text{ UNION } P2]]_{D, NG} = [[P1]]_{D, NG} \cup [[P2]]_{D, NG}$
- f) $[[\text{GRAPH } v (P1)]]_{D, NG} = \langle (v, IRI1) \rangle \bowtie [[P1]]_{D(IRI1), NG} \text{ UNION } \dots \text{ UNION } \langle (v, IRIin) \rangle \bowtie [[P1]]_{D(IRIn), NG}$
- g) $[[\text{GRAPH } IRI(P1)]]_{D, NG} = [[P1]]_{D(IRI), NG}$, if $D(IRI) \in NG$, otherwise $\langle \rangle$.
- h) $[[P1 \text{ FILTER } R]]_{D, NG} = \langle \omega \mid \omega \in [[P1]]_{D, NG} \wedge \text{Eval}_R(\omega) \rangle$

In comparison to the original definition $[[P1 \text{ OPT } P2]]_D = [[P1]]_D \bowtie [[P2]]_D$ given in Pérez et al. (2006), in the case d) of Definition 8 we first join the results of $P1$ and that of $P2$, because the filter expressions in $P2$ may use variables, which are bound in $P1$ rather than in $P2$, for example, $((?x <a> ?z) \text{ AND } (?x ?w)) \text{ OPT } ((?y <a> ?z2) \text{ AND } (?y ?w2) \text{ FILTER } (?z=?z2 \ \&\& \ ?w2 < ?w))$. Without first joining the results of $P1$ and of $P2$, these filter expressions would be evaluated to a different result from the intended one following the SPARQL specification (Prud'hommeaux and Seaborne 2008) and test cases (Feigenbaum 2008). Furthermore, to support named RDF graphs, we have to not only use the default graph D as in Pérez et al. (2006) in our formulas, but also the set of named RDF graphs NG .

For completeness, we have also to define the result of the extended list of graph patterns as well as the query heads:

Definition 9 (Result of query heads and extended list of graph patterns). Let D be an RDF graph and $NG = \{D(IRI1), \dots, D(IRIn)\}$ be a set of named graphs, where $IRI1, \dots, IRIn$ are the IRI labels of the named RDF graphs, $\{v1, \dots, vn\}$ be a set of variables, $R1, \dots, Rn$ be built-in conditions, $\text{ORDER}1, \dots, \text{ORDER}n \in \{\text{ASC}, \text{DESC}\}$, and $\{(s_1, p_1, o_1), \dots, (s_n, p_n, o_n)\}$ bet a set of triple patterns, then:

- (a) $[[\prod_{i \in \{v_1, \dots, v_n\}} P_i]]_{D, NG} = \langle \{(v_1, t_1), \dots, (v_n, t_n)\} / \omega \in [[P_i]]_{D, NG} \wedge (v_i, t_i) \in \omega \text{ for } i \in \{1, \dots, n\} \rangle$.
- (b) $[[DISTINCT P]]_{D, NG} = \langle \omega / \omega \in \{\omega' / \omega' \in [[P]]_{D, NG}\} \rangle$.
- (c) $[[SORT_{ORDER_1(R_1), \dots, ORDER_n(R_n)} P]]_{D, NG} = \langle \omega_1, \dots, \omega_m \rangle$, where $[[P]]_{D, NG} \subseteq \langle \omega_1, \dots, \omega_m \rangle$ and $\langle \omega_1, \dots, \omega_m \rangle \subseteq [[P]]_{D, NG}$, and $\omega_1 \leq ORDER_1(R_1), \dots, ORDER_n(R_n) \omega_2 \leq ORDER_1(R_1), \dots, ORDER_n(R_n) \dots \leq ORDER_1(R_1), \dots, ORDER_n(R_n) \omega_m$, where $\leq ORDER_1(R_1), \dots, ORDER_n(R_n)$ is the order relation between two solutions according to $ORDER_1(R_1), \dots, ORDER_n(R_n)$, that is, $\omega_1 \leq ORDER_1(R_1), \dots, ORDER_n(R_n) \omega_2$ holds if $Value_{R_1}(\omega_1) op_1 Value_{R_1}(\omega_2)$, or $Value_{R_1}(\omega_1) = Value_{R_1}(\omega_2)$ and $Value_{R_2}(\omega_1) op_2 Value_{R_2}(\omega_2)$, ..., or $Value_{R_n}(\omega_1) op_n Value_{R_n}(\omega_2)$, where for each $i \in \{1, \dots, n\}$ op_i is $<$ for $ORDER_i=ASC$ and otherwise op_i is $>$.
- (d) $[[LIMIT i P]]_{D, NG} = \langle \omega_1, \dots, \omega_k \rangle$, where $[[P]]_{D, NG} = \langle \omega_1, \dots, \omega_m \rangle$ and $k = \min(i, m)$.
- (e) $[[OFFSET i P]]_{D, NG} = \langle \omega_i, \dots, \omega_m \rangle$, where $[[P]]_{D, NG} = \langle \omega_1, \dots, \omega_m \rangle$. If $m < i$, then $[[OFFSET i P]]_{D, NG} = \langle \rangle$.
- (f) $[[ASK P]]_{D, NG} = \text{true}$ if $[[P]]_{D, NG} \neq \langle \rangle$ and otherwise false.
- (g) $[[CONSTRUCT\{(s_1, p_1, o_1), \dots, (s_n, p_n, o_n)\} P]]_{D, NG} = \{(s, p, o) / i \in \{1, \dots, n\} \wedge \omega \in [[P]]_{D, NG} \wedge ((s=s_i \wedge s_i \in IUBUL) \vee (s_i, s) \in \omega) \wedge ((p = p_i \wedge p_i \in IUBUL) \vee (p_i, p) \in \omega) \wedge ((o=o_i \wedge o_i \in IUBUL) \vee (o_i, o) \in \omega)\}$.

Example 2 (Result of graph patterns). The result of

$$DISTINCT \prod_{\{?title, ?price\}} ((?x \text{ dc:title } ?title) OPT ((?x \text{ ns:price } ?price) FILTER (?price < 30)))$$

with default RDF graph

$$D = \{ (:book1 \text{ dc:title } "SPARQL Tutorial"), (:book1 \text{ ns:price } 42), (:book2 \text{ dc:title } "The Semantic Web"), (:book2 \text{ ns:price } 23) \}$$

is

$$\begin{aligned} & DISTINCT \prod_{\{?title, ?price\}} ([[(?x \text{ dc:title } ?title)]])_{D, \{ \}} \bowtie \\ & \quad \langle \omega \mid \omega \in [[(?x \text{ dc:title } ?title)]])_{D, \{ \}} \bowtie [[(?x \text{ ns:price } ?price)]])_{D, \{ \}} \\ & \quad \wedge Eval_{price < 30}(\omega) \rangle \\ & = DISTINCT \prod_{\{?title, ?price\}} (\langle \{ (x, :book1), (title, "SPARQL Tutorial"), \\ & \quad \{ (x, :book2), (title, "The Semantic Web") \} \rangle \bowtie \\ & \quad \langle \omega \mid \omega \in \langle \{ (x, :book1), (title, "SPARQL Tutorial"), (price, 42), \\ & \quad \{ (x, :book2), (title, "The Semantic Web"), (price, 23) \} \rangle \\ & \quad \wedge Eval_{price < 30}(\omega) \rangle) \\ & = DISTINCT \prod_{\{?title, ?price\}} (\langle \{ (x, :book1), (title, "SPARQL Tutorial"), \\ & \quad \{ (x, :book2), (title, "The Semantic Web") \} \rangle \bowtie \\ & \quad \langle \{ (x, :book2), (title, "The Semantic Web"), (price, 23) \} \rangle) \\ & = DISTINCT \prod_{\{?title, ?price\}} (\langle \{ (x, :book1), (title, "SPARQL Tutorial"), \\ & \quad \{ (x, :book2), (title, "The Semantic Web"), (price, 23) \} \rangle) \\ & = DISTINCT (\langle \{ (title, "SPARQL Tutorial"), \\ & \quad \{ (title, "The Semantic Web"), (price, 23) \} \rangle) \\ & = \langle \{ (title, "SPARQL Tutorial"), \\ & \quad \{ (title, "The Semantic Web"), (price, 23) \} \rangle \rangle \end{aligned}$$

5.2 Logical Optimization Rules

Logical optimization aims to reorganize the operator graph into an equivalent operator graph, which generates the same output for any input as the original one, in order to optimize the execution time of query evaluation. We write $P1 \equiv P2$ to denote that a graph pattern $P1$ generates the same output for any input as the graph pattern $P2$.

Figure 5.1 presents some equivalency rules published in the contributions (Pérez et al. 2006; Groppe et al. 2007a, b; Heese et al. 2006), which can be used for the logical optimization of the operator graph of the SPARQL query. We will explain some important logical optimization rules in detail. Many of these equivalency rules are adapted from the equivalency rules of the relational algebra (see e.g., Arasu et al. 2006; Chaudhuri 1998; Ioannidis 1996; Jarke and Koch 1984). However, to the best of our knowledge, the equivalency rules for pushing filters and constant and variable propagations have been only informally described and have not been so precisely formalized and comprehensively presented as here.

A description on the logical optimization rules developed in our LUPOSDATE system can be accessed via http://www.ifis.uni-luebeck.de/~groppe/tutorial_demo/ruledoc/.

5.2.1 Pushing FILTER Operators

It is a good strategy to push *FILTER* operators as much as possible into inner subexpressions. Early application of *FILTER* operators will reduce the sizes of intermediate results and thus speed up succeeding processing.

To define the conditions under which a *FILTER* operator can be moved into inner subexpressions, we must define the function $nov(P)$, where P is a graph pattern. $nov(P)$ determines a set of variables, which are bound in every result solution of P for any input. For example, $nov((?book \text{ dc:author } ?author) \text{ UNION } (?book \text{ dc:price } ?price)) = \{?book\}$, as every result solution binds $?book$ with a value regardless if the result solution is in the result of $(?book \text{ dc:author } ?author)$ or of $(?book \text{ dc:price } ?price)$. However, the variables $?author$ and $?price$ might not be bound.

Definition 10. Let $\{\}$ be the empty graph pattern, $(s \text{ p } o) \in (I \cup B \cup L \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$ be a triple pattern, $P1$ and $P2$ be two graph patterns, and R be a built-in condition. The function $nov(P)$ is recursively defined as follows:

- $nov(\{\}) = \{\}$
- $nov((s \text{ p } o)) = \{v \mid v \in \{s, p, o\} \wedge v \in V\}$
- $nov(P1 \text{ AND } P2) = nov(P1) \cup nov(P2)$
- $nov(P1 \text{ UNION } P2) = nov(P1) \cap nov(P2)$
- $nov(P1 \text{ OPT } P2) = nov(P1)$
- $nov(P1 \text{ FILTER } R) = nov(P1)$

Equivalency Rules	Reference
<p>AND and UNION are associative and commutative, i.e.,</p> $(P1 \text{ AND } P2) \equiv (P2 \text{ AND } P1)$ $(P1 \text{ UNION } P2) \equiv (P2 \text{ UNION } P1)$ $(P1 \text{ AND } (P2 \text{ AND } P3)) \equiv ((P1 \text{ AND } P2) \text{ AND } P3)$ $(P1 \text{ UNION } (P2 \text{ UNION } P3)) \equiv ((P1 \text{ UNION } P2) \text{ UNION } P3)$	Pérez et al. (2006)
$(P1 \text{ AND } (P2 \text{ UNION } P3)) \equiv ((P1 \text{ AND } P2) \text{ UNION } (P1 \text{ AND } P3))$ $(P1 \text{ OPT } (P2 \text{ UNION } P3)) \equiv ((P1 \text{ OPT } P2) \text{ UNION } (P1 \text{ OPT } P3))$ $((P1 \text{ UNION } P2) \text{ OPT } P3) \equiv ((P1 \text{ OPT } P3) \text{ UNION } (P2 \text{ OPT } P3))$ $((P1 \text{ UNION } P2) \text{ FILTER } R) \equiv ((P1 \text{ FILTER } R) \text{ UNION } (P2 \text{ FILTER } R))$	Pérez et al. (2006)
<p>$P = (P1 \text{ AND } (P2 \text{ OPT } P3)) \equiv ((P1 \text{ AND } P2) \text{ OPT } P3)$, where P is a well designed graph pattern, i.e. for every occurrence of a sub-pattern $P' = (P1' \text{ OPT } P2')$ of P and for every variable ?X occurring in P, it is required that if ?X occurs both inside P2' and outside P', then it also occurs in P1'.</p>	Pérez et al. (2006)
<p>$P = ((P1 \text{ OPT } P2) \text{ OPT } P3) \equiv ((P1 \text{ OPT } P3) \text{ OPT } P2)$, where P is a well designed graph pattern</p>	Pérez et al. (2006)
<p><i>Pushing filter upward in the operatorgraph:</i> We can push filter upward until after those operators in the operatorgraph, where <i>all</i> variables of the filter expression have already been bound. This reduces the space used for intermediary results.</p>	Groppe et al. (2007)
<p><i>Filtering unsatisfiable queries and subexpressions:</i> A query or a subexpression of it is unsatisfiable if it returns the empty result for any RDF data. Unsatisfiability can be often precomputed and be used to simplify queries. For example, the predicate $\text{FILTER}(\text{?price} < 30 \ \&\& \ \text{?price} > 50)$ is unsatisfiable, since $\text{?price} < 30$ is contradictory to $\text{?price} > 50$. We can hence replace the filter expression with $\text{FILTER}(\text{false})$.</p>	Some rules in Heese (2006)
<p><i>Eliminating Variables:</i> For each $\text{Filter}(\text{?var1} = \text{?var2})$ and ?var2 is not bound in an outer scope of $\text{Filter}(\text{?var1} = \text{?var2})$, we can replace the variable ?var2 with ?var1 and eliminate the filter $\text{Filter}(\text{?var1} = \text{?var2})$.^a</p>	Heese (2006)

Fig. 5.1 Equivalency rules for logical optimization of SPARQL queries

^aRemark: If the results of the variables are typed-data, e.g. 1 and +1, two results are value-equal, but literally un-identical. These situations might occur when the variables do not occur as the subject or the predicate of a triple pattern, but only as the objects, which may be bound with numerical typed literals or language tagged literals. The equivalency rule cannot be applied in these cases.

Furthermore, let PR be a given built-in condition or graph pattern, and let $\text{var}(\text{PR})$ be a function, which returns all variables occurring in PR. The following equivalency rules can be used to move a FILTER operator into inner subexpressions:

- $(P1 \text{ AND } P2) \text{ FILTER } R \equiv (P1 \text{ FILTER } R) \text{ AND } (P2 \text{ FILTER } R)$ if $\text{var}(R) \subseteq \text{nov}(P1) \wedge \text{var}(R) \subseteq \text{nov}(P2)$
- $(P1 \text{ AND } P2) \text{ FILTER } R \equiv (P1 \text{ FILTER } R) \text{ AND } P2$ if $\text{var}(R) \subseteq \text{nov}(P1) \wedge \text{var}(R) \not\subseteq \text{nov}(P2)$
- $(P1 \text{ AND } P2) \text{ FILTER } R \equiv P1 \text{ AND } (P2 \text{ FILTER } R)$ if $\text{var}(R) \subseteq \text{nov}(P2) \wedge \text{var}(R) \not\subseteq \text{nov}(P1)$
- $(P1 \text{ OPT } P2) \text{ FILTER } R \equiv (P1 \text{ FILTER } R) \text{ OPT } (P2 \text{ FILTER } R)$ if $\text{var}(R) \subseteq \text{nov}(P1) \wedge \text{var}(R) \subseteq \text{nov}(P2)$
- $(P1 \text{ OPT } P2) \text{ FILTER } R \equiv (P1 \text{ FILTER } R) \text{ OPT } P2 \text{ FILTER } R$ if $\text{var}(R) \subseteq \text{nov}(P1) \wedge \text{var}(R) \not\subseteq \text{nov}(P2)$
- $(P1 \text{ UNION } P2) \text{ FILTER } R \equiv (P1 \text{ FILTER } R) \text{ UNION } (P2 \text{ FILTER } R)$

5.2.2 Splitting and Commutativity of FILTER Operators

Let $R1$ and $R2$ be two built-in conditions. The simple equivalency rule $\text{FILTER}(R1 \wedge R2) \equiv \text{FILTER}(R1) \text{ FILTER}(R2)$ splits a FILTER operator with a built-in condition $R1 \wedge R2$ into two separate FILTER operators, which can be further optimized according to other equivalency rules discussed in the following subsections.

For example, the filter in $\text{FILTER}(?author=?editor \wedge ?price < 30) ((?book \text{ dc:author } ?author) \text{ AND } (?book \text{ dc:editor } ?editor) \text{ AND } (?book \text{ dc:price } ?price))$ cannot be moved to the inner subexpressions without splitting. However, after splitting the filter operator, that is, $\text{FILTER}(?author=?editor) \text{ FILTER}(?price < 30)$, the filter operator $\text{FILTER}(?price < 30)$ can be moved to the triple pattern $(?book \text{ dc:price } ?price)$, that is, $\text{FILTER}(?author=?editor) ((?book \text{ dc:author } ?author) \text{ AND } (?book \text{ dc:editor } ?editor) \text{ AND } (\text{FILTER}(?price < 30) (?book \text{ dc:price } ?price)))$.

Furthermore, $\text{FILTER}(R1) \text{ FILTER}(R2) \equiv \text{FILTER}(R2) \text{ FILTER}(R1)$ holds; that is, the FILTER operator is commutative.

5.2.3 Constant and Variable Propagation

In the graph pattern $P = ((?person \text{ foaf:name } ?name) \text{ FILTER}(?name = \text{“Bob”}))$, we can replace the variable $?name$ in the triple pattern $(?person, \text{foaf:name}, ?name)$ with the constant string “Bob” in the filter expression. Consequently, the filter expression becomes redundant, and P can be optimized to $P' = (?person \text{ foaf:name } \text{“Bob”})$. However, the result of P contains the bindings of the variables $?person$ and $?name$, but the result of P' consists only of the binding of the variable $?person$.

In order to solve this problem, we need a new operation $\text{BIND}(?name = \text{“Bob”})$ in P' , which binds the variable $?name$ with “Bob” in the result of P' . Thus, $P' = ((?person \text{ foaf:name } \text{“Bob”}) \text{ BIND}(?name = \text{“Bob”}))$ returns the same result as P for any input RDF graph. This kind of equivalency is also called constant propagation.

Typed data, for example, 1 and +1, are value-equal, but not identical. Consequently, for example, the triple patterns $(?s ?p 1)$ and $(?s ?p +1)$ return different results, but $(?s ?p ?o) \text{ FILTER}(?o=1)$ compares $?o$ to be value-equal to 1; that is, $[[(?s ?p 1) \text{ UNION } (?s ?p +1)]]_{D, NG} \subseteq [[\prod_{\{?s, ?p\}} ((?s ?p ?o) \text{ FILTER}(?o=1))]]_{D, NG}$ for any default graph D and for any named graphs NG . Similar remarks apply to language-tagged literals as well. Therefore, the constant propagation cannot be applied for typed data, where non-identical values can be value-equal like in the case of numerical values, or language-tagged literals.

Without application of optimization rules, the graph pattern $((?person \text{ foaf:name } ?name) \text{ AND } (?person2 \text{ foaf:mbox } ?mbox) \text{ FILTER}(?person2 = ?person))$ will lead to an inefficient query execution: a Cartesian product between the two triple patterns is first computed, and afterward the *FILTER* operation is applied.

However, we can leverage the information $?person2 = ?person$ in the filter expression to rewrite the graph pattern to $((?person \text{ foaf:name } ?name) \text{ AND } (?person \text{ foaf:mbox } ?mbox) \text{ BIND}(?person2 = ?person))$. The *BIND*($?person2 = ?person$) operation ensures that the result also contains the bindings of the replaced variable. Consequently, the costly Cartesian product is replaced with a join.

However, $\langle \{(?person=1)\} \rangle \bowtie \langle \{(?person=+1)\} \rangle = \langle \rangle$, but $\langle \{(?person=1, ?person2=+1)\} \rangle$ remains after *FILTER*($?person2 = ?person$) has been applied to $\langle \{(?person=1, ?person2=+1)\} \rangle$, as 1 and +1 are value-equal and not identical. Therefore, the rule is incorrect whenever the variables $?person$ and $?person2$ can contain value-equal (but not identical) values, which occur, for example, for numerical or language-tagged literals. Numerical or language-tagged literals can only occur in objects of RDF triples. Therefore, if one of the variables $?person$ or $?person2$ occurs as subject or predicate in one of the triple patterns to be joined, then the variables $?person$ and $?person2$ cannot contain value-equal (but not identical) values, and the rule is correct.

Let P be a graph pattern, $v \in V$ a variable, and $cv \in LU \cup V$ a constant value or another variable. The expression $P \text{ BIND}(v=cv)$ adds a binding (v, cv) to each result solution of the graph pattern P . Let D be the input RDF graph and NG be the named graphs. $[[P \text{ BIND}(v=cv)]]_{D, NG}$ is formally defined as follows:

$$[[P \text{ BIND}(v=cv)]]_{D, NG} = \langle E / E' \in [[P]]_{D, NG} \wedge E = E' \cup \{(v, v') \mid (v'=cv \wedge cv \in LU) \vee ((cv, v') \in E' \wedge cv \in V)\} \rangle.$$

For each $i \in \{1, \dots, n\}$ let $(s_i p_i o_i) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$ be a triple pattern, and $cv \in (I \cup L \cup V)$. Our logical optimization rule to constant and variable propagation can be expressed as follows:

$$(s_1 p_1 o_1) \text{ AND } \dots \text{ AND } (s_n p_n o_n) \text{ FILTER}(v=cv) \equiv (s_1' p_1' o_1') \text{ AND } \dots \text{ AND } (s_n' p_n' o_n') \text{ BIND}(v=cv) \text{ if } v \in \{s_i, p_i, o_i\} \text{ and } cv \text{ is neither a numerical value nor a language-tagged literal nor a variable with } \forall i: s_i \neq v \wedge p_i \neq v \wedge s_i \neq cv \wedge p_i \neq cv, \text{ where } \forall i: s_i' = cv \text{ if } s_i = v; \text{ otherwise } s_i' = s_i, p_i' = cv \text{ if } p_i = v; \text{ otherwise } p_i' = p_i, \text{ and } o_i' = cv \text{ if } o_i = v; \text{ otherwise } o_i' = o_i.$$

BIND($v=cv$) adds an additional variable binding to the query result. Therefore, the performance improves when *BIND*($v=cv$) can be moved to outer subexpressions, such that inner subexpressions have smaller intermediate results. Let PI and

$P2$ be graph patterns, and R be a built-in condition, then $BIND(v=cv)$ can be moved to outer subexpressions according to the following equivalency rules:

- $(P1\ BIND(v=cv))\ AND\ P2 \equiv (P1\ AND\ P2)\ BIND(v=cv)$ if $v \notin var(P2) \wedge cv \notin var(P2)$
- $P1\ AND\ (P2\ BIND(v=cv)) \equiv (P1\ AND\ P2)\ BIND(v=cv)$ if $v \notin var(P1) \wedge cv \notin var(P1)$
- $(P1\ BIND(v=cv))\ OPT\ P2 \equiv (P1\ OPT\ P2)\ BIND(v=cv)$ if $v \notin var(P2) \wedge cv \notin var(P2)$
- $(P1\ BIND(v=cv))\ FILTER\ R \equiv (P1\ FILTER\ R)\ BIND(v=cv)$ if $v \notin var(R)$
- $(P1\ BIND(v=cv))\ OP\ (P2\ BIND(v=cv)) \equiv (P1\ OP\ P2)\ BIND(v=cv)$, where $OP \in \{AND, UNION, OPT\}$

5.2.4 Heuristic Query Optimization Using Equivalency Rules

Query optimizers apply an equivalency rule only when its application can lead to a better performance. However, the difficult task for query optimizers is to *decide* if an equivalency rule can lead to a better performance. Often, logical optimization approaches use heuristic methods. Heuristic methods assume that the application of certain equivalency rules typically lead to a better performance and thus always apply these equivalency rules in any possible cases.

While heuristic methods may lead to suboptimal results, it is a simple approach, and it is especially favourite to some equivalency rules like pushing filter operators into inner subexpressions. The application of the rule can early filter intermediate results and thus improving the performance in nearly every case. We already obtain a good logical operator graph when we just apply the equivalency rules of the previous subsections for splitting *FILTER* operations, pushing *FILTER* operators into inner subexpressions, and constant and variable propagations. This heuristic approach is used by most query optimizers in practice in combination with other query optimization approaches to enumerate all possible join orderings as we will later discuss. Figure 5.2 presents an example of the heuristic approach to optimizing queries using equivalency rules.

For many other equivalency rules, especially those for changing the join order, logical optimizers typically use statistics of the input data and a cost model to determine the best logical operator graph. For example, in order to determine the best join order, the query optimizer enumerates all possible orderings using dynamic programming, estimates the costs of each possibility using statistics and the cost model and chooses the best estimated one as optimized logical operator graph.

The next subsection describes cost-based optimizations, that is, optimizations, which determine estimated costs of a query plan and aim to choose the estimated best ones. Afterward, we introduce *histograms*, which are used in statistic-based approaches to cost estimation of logical and physical operators.

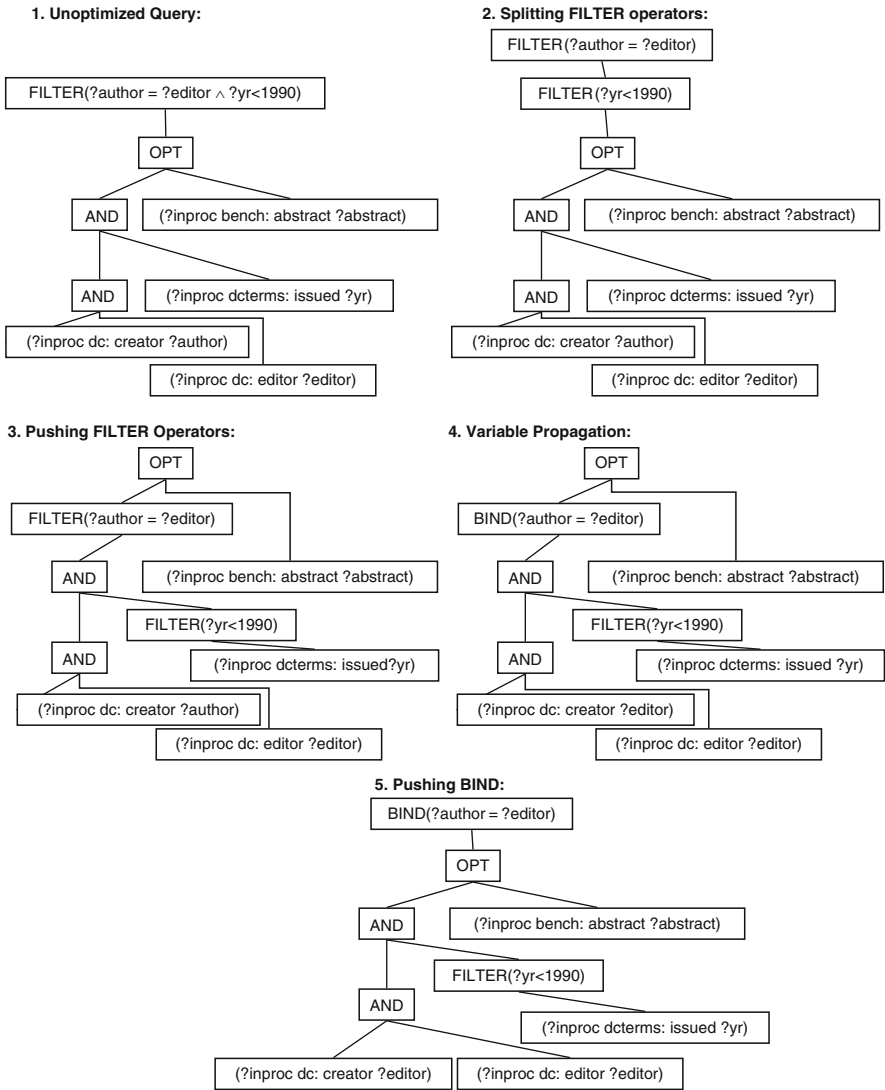


Fig. 5.2 Heuristic approach to optimizing queries using equivalency rules

5.2.5 Cost-Based Optimization

Cost-based optimization aims to optimize a whole query plan containing operators such as join, filter, projection, sort, and union by estimating their costs regarding consumed space in memory and on disk, and processing time. Due to the fact that the already discussed heuristic approaches like pushing filters and variable and constant propagations optimize the query plan in nearly every case, query

optimizers in practice often apply these heuristic approaches and only apply further cost-based optimizations as discussed in this subsection for join (and optional) order optimization. This has the advantage that less possibilities for query plans need to be considered and thus query optimization becomes faster.

Join orders determine the sizes of intermediate results (e.g., DeHaan and Tompa 2007), and are an important factor, which influences the performance of query processing. Therefore, a good execution plan depends much on join ordering. We focus on join order optimization in our examples and introduce special join order optimization approaches besides approaches to optimize the whole query plan.

In the first subsection, we describe the heuristic approaches to join ordering optimization suitable for main-memory databases. These approaches can be performed fast, such that they are not slower than processing a not optimized join ordering, but may produce suboptimal results, the performance lost of which in comparison to the optimal join ordering can be neglected for the small datasets of main-memory databases. In the second subsection, we describe the approaches, which enumerate all possible query plans and choose the estimated best query plan from them. These approaches spend much more time on query optimization, but produce better query plans. Thus, they are the first choice for large-scale datasets, where processing a suboptimal query plan becomes much more expensive, that is, are slower processed, than for small datasets.

For join order optimization, we must have in mind that certain join algorithms, which we introduce in the next chapter, like merge join have lower costs than others like hash join or even nested loop join, such that not only the estimated cardinality must be considered for join optimization, but also if cheap join algorithms like the merge join can be used for a considered join.

Furthermore, the different optimization approaches generate different types of join trees. In left-deep join trees (see Fig. 5.3a), all right children of joins are triple patterns. Contrary, in right-deep join trees (see Fig. 5.3c), all left children of joins are triple patterns. All other join trees are bushy join trees (see Fig. 5.3b). Whereas left-deep and right-deep join trees allow using indices on the input data in each join for the triple pattern operand, bushy trees are more flat having more possibilities to optimize the join order and allowing better parallel strategies.

5.2.5.1 Heuristic Approaches to Join Order Optimization

The first simple approach generates left-deep join trees. Note that some join algorithms like our main-memory join approach described in the next chapter work only on left-deep join trees (or on right-deep join trees) rather than on bushy join trees. A left-deep (or right-deep) join tree can be generated by first choosing two triple patterns to join and afterward choosing a triple pattern to the already generated join so far (until no triple pattern remains).

An important impact on the determination of execution order of triple patterns is the restrictiveness of triple patterns: a more restrictive triple pattern typically retrieves less data than the less restrictive triple patterns.

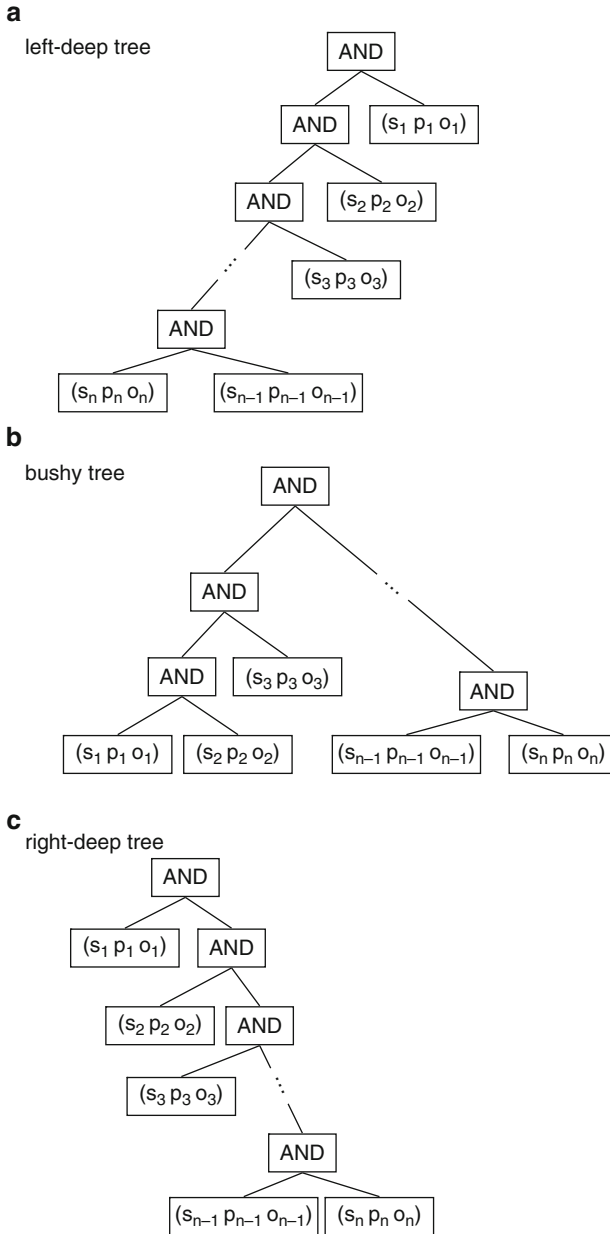


Fig. 5.3 Left-deep, bushy, and right-deep join trees

Definition 11 (more restrictive triple patterns). Let $tp_1, tp_2 \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$ be two triple patterns. If tp_1 contains less number of variables than tp_2 , then tp_1 is more restrictive than tp_2 .

Let $\text{var}(P)$ be the function, which returns the variables occurring in a graph pattern P . Furthermore, let P_{old} be the graph pattern of already joined triple patterns and tp be a remaining triple pattern. Then we call variables in $\text{var}(P_{\text{old}}) \cap \text{var}(tp)$ old variables of tp , because they are already bound in P_{old} during processing. We call all other variables of tp , that is, $\text{var}(tp) - (\text{var}(P_{\text{old}}) \cap \text{var}(tp))$, new variables of tp , because they have not been bound so far in P_{old} . According to our main-memory join approach in the next chapter, the restrictiveness of a triple pattern is determined by the number of new variables in the triple pattern. Therefore, we first choose the most restrictive triple pattern, which contains the most constant values. Afterward, we choose that triple pattern tp , which binds the least new variables, to be joined with the other already joined triple patterns; that is, we build $P_{\text{old}} \text{ AND } tp$, until no triple pattern remains. In this way, join operands can be reordered according to the restrictiveness of triple patterns without considering the input data.

Example 3 (Reordering triple patterns according to the restrictiveness in order to optimize join computation). The following triple patterns

<i>(?article, rdf:type, bench:Article)</i>	<i>AND</i>
<i>(?article, dc:creator, ?person)</i>	<i>AND</i>
<i>(?inproc, rdf:type, bench:Inproceedings)</i>	<i>AND</i>
<i>(?inproc, dc:creator, ?person2)</i>	<i>AND</i>
<i>(?person, foaf:name, ?name)</i>	<i>AND</i>
<i>(?person2, foaf:name, ?name2)</i>	

can be reordered as follows using our heuristic approach choosing the triple pattern with least new variables:

(A)	<i>(?inproc, rdf:type, bench:Inproceedings)</i>	<i>AND</i>
(B)	<i>(?inproc, dc:creator, ?person2)</i>	<i>AND</i>
(C)	<i>(?person2, foaf:name, ?name2)</i>	<i>AND</i>
(D)	<i>(?article, rdf:type, bench:Article)</i>	<i>AND</i>
(E)	<i>(?article, dc:creator, ?person)</i>	<i>AND</i>
(F)	<i>(?person, foaf:name, ?name)</i>	

After reordering in our example, each triple pattern contains only one new variable, which is not already bound by a previous triple pattern before. Other reorderings are also possible, where each triple pattern contains only one new variable, for example, the reordering (D), (A), (E), (B), (C), and (F).

Triple patterns can also be reordered according to their result sizes or according to the estimated join cardinality using, for example, histograms. Result sizes of triple patterns can be determined efficiently using special indices over the input data. If we do not consider already bound variables, we can determine the result sizes of each triple pattern with one index access. However, our approach to main-memory join computation described in the next chapter replaces already bound variables. Therefore, ordering the join operands according to their result sizes without considering already bound variables leads to a suboptimal reordering. Otherwise, more complex approaches for join cardinality estimations must be

applied using, for example, histograms. However, for main-memory databases, datasets are small and approaches for join cardinality estimations are often slower than processing a less optimized join order. Thus, whereas applying approaches for join cardinality estimations for optimizing joins in large datasets is a must for best performance, they typically slow down the performance in main-memory databases and should not be applied there.

In a hybrid approach, we order the triple patterns primarily according to restrictiveness, which we call the main factor, and secondarily according to result sizes, which we call the minor factor. Whenever the main factor cannot unambiguously determine the order of the triple patterns, the second factor is used to help reach the determination.

The experimental evaluation in the next chapter shows that SPARQL main-memory evaluation performs best when the restrictiveness is the main factor and the result size is the minor factor. As we explained already before, join ordering according to the result sizes of triple patterns without considering already bound variables leads to a suboptimal order.

5.2.5.2 Enumeration of Plans

The heuristic approaches for join order optimization described so far are preferable for main-memory databases, as they can be computed very fast, but can lead to suboptimal join orderings, which are still fast processed because of small datasets. For large-scale datasets, it makes sense to spend more time on query optimization for computing better query plans, as they can lead to enormous performance improvements. Another class of optimization approaches following this idea aims to enumerate all possible query plans, determines the estimated costs of each possible query plan and chooses the one with the best estimated costs. For more complex queries, enumerating all possible query plans, estimating their costs and choosing the best one becomes too slow because of a blowup of the query plan possibilities. Therefore, the approaches for query plan enumeration try to not consider whole subsets of the plans if these subsets cannot lead to the best estimated costs.

In principle, there are two main strategies for enumerating query plans:

1. Top-down: starting at the root of the logical query plan, each possibility to compute its operands is considered by applying equivalency rules, and the costs of each possibility are computed, where the best one is chosen. Note that the enumeration of plans is recursively applied.
2. Bottom-up: starting at the leafs of a query plan, for each subexpression S of the logical query plan, all possibilities to compute the subexpression S are enumerated based on the equivalency rules and their costs are determined by combining already determined possible query plans for subexpressions of S .

Both kinds of strategies enumerate all possible query plans. However, the bottom-up strategies are more popular, as they can discard the plans with high costs early.

In the following subsections, we describe several important approaches for plan enumeration. The last subsection describes also an example of a query optimizer used in practice, which is actually the query optimizer in our SPARQL engines for large-scale datasets.

Branch and Bound Plan Enumeration

In this approach, first a query plan is determined based on heuristics like pushing filters, variable and constant propagation, and the already discussed heuristic approaches to join order optimization and its costs are estimated. These costs serve now as upper limit for the costs of the other plans and the query plan becomes the currently best one. Afterward, all possible query plans are enumerated using the bottom-up strategy. If the estimated costs of an enumerated subexpression are already higher than the determined upper limit for the costs, then this subexpression can be discarded as all query plans containing this subexpression will have higher costs than the currently best one. If we find a query plan with lower costs, then this query plan becomes the currently best one and these lower costs become the new upper limit. The advantage of this strategy is that the query optimizer can stop after a plan with very low costs is found, such that the time spent for query optimization does not dominate the overall query processing costs. On the other hand, if the currently best query plan has high costs, then it is wise to continue query optimization, such that may be a better query plan is found and the costs are significantly reduced.

Hill Climbing

This approach also first computes a query plan using good heuristics. Afterward, the approach tries to lower the costs of this query plan by making small changes to the query plan. If all possible, small changes do not lower the costs, then the current query plan is chosen. In this way good candidates for low-cost query plans are chosen quickly. Like the branch and bound plan enumeration approach, hill climbing can be stopped when a query plan with very low costs is already found.

However, as big changes to the query plan are *not* made, *not* all possible query plans are enumerated, and hill climbing therefore may miss optimal query plans.

Dynamic Programming

Dynamic programming is a variant of the general bottom-up strategy. Query plans for a complex subexpression S are determined by building every possible disjoint

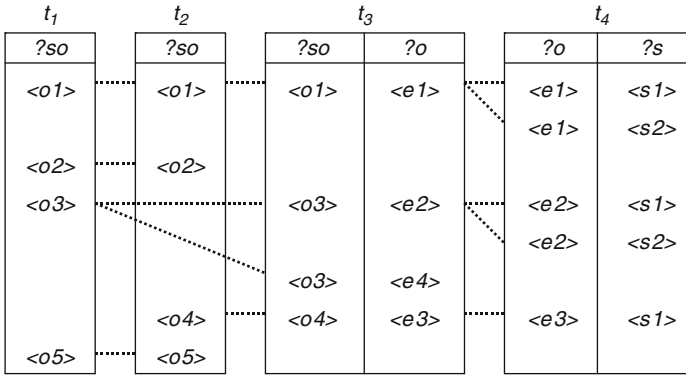


Fig. 5.4 Results of the triple patterns t_1 , t_2 , t_3 , and t_4 in the example, where the dotted lines connect join partners between the different triple patterns

subsets $S1$ and $S2$ of its subexpressions (e.g., sets of triple patterns to be joined), the union of which is S ; that is, $S = S1 \cup S2$. Assuming that we have already determined the best possible query plans for $S1$ and $S2$, we determine the query plan and its costs for the combination of $S1$ and $S2$. Among all the built disjoint subsets $S1$ and $S2$ and their combined query plans for S , we choose the one with the best estimated costs.

We now present how to determine the best query plan for joining four triple patterns t_1 , t_2 , t_3 , and t_4 . For an example, t_1 is $\langle s1 \rangle \langle p1 \rangle ?so$, t_2 is $\langle s2 \rangle \langle p2 \rangle ?so$, t_3 is $?so \langle p3 \rangle ?o$, and t_4 is $?s \langle p4 \rangle ?o$. We assume to have input data such that the result of t_1 is $\langle \{ (?so, \langle o1 \rangle) \}, \{ (?so, \langle o2 \rangle) \}, \{ (?so, \langle o3 \rangle) \}, \{ (?so, \langle o5 \rangle) \} \rangle$, the result of t_2 is $\langle \{ (?so, \langle o1 \rangle) \}, \{ (?so, \langle o2 \rangle) \}, \{ (?so, \langle o4 \rangle) \}, \{ (?so, \langle o5 \rangle) \} \rangle$, the result of t_3 is $\langle \{ (?so, \langle o1 \rangle), (?o, \langle e1 \rangle) \}, \{ (?so, \langle o3 \rangle), (?o, \langle e2 \rangle) \}, \{ (?so, \langle o3 \rangle), (?o, \langle e4 \rangle) \}, \{ (?so, \langle o4 \rangle), (?o, \langle e3 \rangle) \} \rangle$, and the result of t_4 is $\langle \{ (?s, \langle s1 \rangle), (?o, \langle e1 \rangle) \}, \{ (?s, \langle s2 \rangle), (?o, \langle e1 \rangle) \}, \{ (?s, \langle s1 \rangle), (?o, \langle e2 \rangle) \}, \{ (?s, \langle s2 \rangle), (?o, \langle e2 \rangle) \}, \{ (?s, \langle s1 \rangle), (?o, \langle e3 \rangle) \} \rangle$. We present these results of the triple patterns also in Fig. 5.4, where we additionally mark join partners between the results of the triple patterns by dotted lines.

First the costs are estimated of processing each triple pattern. We have not described so far how to estimate the costs of operators, which we will do in the section about histograms. In this section, we do not estimate the costs, but compute the sizes of all intermediate results, that is, the number of solutions in the result, and take this number as costs. Afterward, the minimal costs are determined of a join between any two triple patterns; that is, we determine the following table:

Join between	t_1, t_2	t_1, t_3	t_1, t_4	t_2, t_3	t_2, t_4	t_3, t_4
Determine plan with	$t_1 \bowtie t_2$	$t_1 \bowtie t_3$	$t_1 \bowtie t_4$	$t_2 \bowtie t_3$	$t_2 \bowtie t_4$	$t_3 \bowtie t_4$
minimal costs among	$t_2 \bowtie t_1$	$t_3 \bowtie t_1$	$t_4 \bowtie t_1$	$t_3 \bowtie t_2$	$t_4 \bowtie t_2$	$t_4 \bowtie t_3$
Best plan in example:	$t_1 \bowtie t_2$	$t_1 \bowtie t_3$	$t_1 \bowtie t_4$	$t_2 \bowtie t_3$	$t_2 \bowtie t_4$	$t_3 \bowtie t_4$
Costs:	3	3	20	2	20	5

Note that, for example, $t_1 \bowtie t_2$ and $t_2 \bowtie t_1$ may have different costs for a join algorithm, as the join is faster processed when the cardinality of the left operand is smaller than the one of the right operand (see next chapter). Note that in our example, this does not matter because we determine the costs by looking only at the number of all intermediate results.

We now define the function $min(T)$, where $T \subseteq \{t_1, t_2, t_3, t_4\}$ is a set of triple patterns, to return the already determined query plan with minimal costs for joining the triple patterns of T by looking up the previous table. Then we compute the following table for joins between three triple patterns:

Join between	t_1, t_2, t_3	t_1, t_3, t_4	t_2, t_3, t_4
Determine plan with minimal costs among	$min(\{t_1, t_2\}) \bowtie t_3$ $t_3 \bowtie min(\{t_1, t_2\})$ $min(\{t_1, t_3\}) \bowtie t_2$ $t_2 \bowtie min(\{t_1, t_3\})$ $min(\{t_2, t_3\}) \bowtie t_1$ $t_1 \bowtie min(\{t_2, t_3\})$	$min(\{t_1, t_3\}) \bowtie t_4$ $t_4 \bowtie min(\{t_1, t_3\})$ $min(\{t_1, t_4\}) \bowtie t_3$ $t_3 \bowtie min(\{t_1, t_4\})$ $min(\{t_3, t_4\}) \bowtie t_1$ $t_1 \bowtie min(\{t_3, t_4\})$	$min(\{t_2, t_3\}) \bowtie t_4$ $t_4 \bowtie min(\{t_2, t_3\})$ $min(\{t_2, t_4\}) \bowtie t_3$ $t_3 \bowtie min(\{t_2, t_4\})$ $min(\{t_3, t_4\}) \bowtie t_2$ $t_2 \bowtie min(\{t_3, t_4\})$
Best plan in example: Costs:	$2 + 1 = 3$	$5 + 4 = 9$	$5 + 3 = 8$

At last, we compute the query plan with minimal costs for the four triple patterns, which is our final optimized query plan:

Join between	t_1, t_2, t_3, t_4
Determine plan with minimal costs among	$min(\{t_1, t_2, t_3\}) \bowtie t_4$ $t_4 \bowtie min(\{t_1, t_2, t_3\})$ $min(\{t_1, t_2, t_4\}) \bowtie t_3$ $t_3 \bowtie min(\{t_1, t_2, t_4\})$ $min(\{t_1, t_3, t_4\}) \bowtie t_2$ $t_2 \bowtie min(\{t_1, t_3, t_4\})$ $min(\{t_2, t_3, t_4\}) \bowtie t_1$ $t_1 \bowtie min(\{t_2, t_3, t_4\})$ $min(\{t_1, t_2\}) \bowtie min(\{t_3, t_4\})$ $min(\{t_3, t_4\}) \bowtie min(\{t_1, t_2\})$ $min(\{t_1, t_3\}) \bowtie min(\{t_2, t_4\})$ $min(\{t_2, t_4\}) \bowtie min(\{t_1, t_3\})$ $min(\{t_1, t_4\}) \bowtie min(\{t_2, t_3\})$ $min(\{t_2, t_3\}) \bowtie min(\{t_1, t_4\})$
Best plan in example: Costs:	$min(\{t_1, t_2, t_3\}) \bowtie t_4$ $3 + 2 = 5$

In our example, we therefore have determined the join order $((t_1 \bowtie t_2) \bowtie t_3) \bowtie t_4$ as optimal join order.

Selinger-Style Optimization

Dynamic programming yields often the best query plan, but not always: For example, a query plan A with high costs produces *sorted* results, but another

query plan B for the same subexpression with lower costs does not. If now the sorting of the result can be used in a more complex subexpression S to apply (one or even several times) an algorithm with lower costs, then the overall costs may be smaller than the ones when using the query plan B and an algorithm with higher costs in S . The algorithm with lower costs in S requiring sorted input may be merge joins, optimized versions of duplicate elimination or group-by operators, or a sort operator may be discarded. In order to consider these cases, the Selinger-style optimization (Griffiths-Selinger et al. 1979) has been developed as a variant of dynamic programming.

In comparison to dynamic programming, not only the best query plan for a subexpression is kept for the Selinger-style optimization, but also the best query plans, the evaluation of which produce sorted results. These query plans are additionally considered when the query plans for more complex subexpressions are built. Query optimizers of professional database management systems often apply internally this Selinger-style optimization.

Example of a Query Optimizer Used in Practice

In this section, we describe our implemented SPARQL query optimizer in LUPOSDATE for large-scale datasets as example of a query optimizer used in practice. We employ the technique of dynamic programming for generating the execution plan with the optimal join order. Our query optimizer uses equi-depth histograms (Piatetsky-Shapiro and Connell 1984) of triple patterns for calculating the cost of individual joins and the overall cost of a concrete plan.

Since merge joins without additional sorting phases are very cheap, our query optimizer first chooses such merge joins. When data become unsorted, we use the merge join together with our fast sorting technique, which we describe in the next chapter. If a query contains many triple patterns, we split the set of triple patterns into two sets according to either Cartesian products (first choice), membership of the largest star-shaped joins (second choice), or paths (third choice). The join orders of these subsets are then optimized separately. If there is only one huge star-shaped join or one huge path join, this join is optimized greedy by first joining those triple patterns, the results of which have the smallest cardinality. This strategy scales well for a number of triple patterns and can generate near optimal plans. In our experiments, we have used this strategy for more than seven triple patterns, which actually occurs very seldom in real-world queries.

Whenever a bloom filter needs to be computed from the results of one join operand, our query optimizer plans the computation of this join operand more early than the other join operand. Consequently, index scans can use this bloom filter to filter out irrelevant data for the other join operand more early. If bloom filters are computed from both join operands, then the join operand with the lower cost is computed first. This bloom filter from the operand with the lower cost can be used to reduce the high cost of the other operand by earlier filtering out irrelevant data of it.

5.2.6 Histograms

Histograms describe the properties of the input data and are the basis for good estimations of result sizes of operations. Since a histogram is used for cost estimations, it does not matter if the histogram does not reflect recent database updates, especially when the recent updates do not change much the properties of the input data. Therefore, a histogram does not need to be recomputed after every database modification, but is recomputed in (automatic and/or manual) statistics-gathering cycles.

The most common types of histograms (Piatetsky-Shapiro and Connell 1984) are equal-width, equal-depth, and most-frequent-values histograms. The most-frequent-values histogram can be combined with the equal-width and equal-depth histograms.

By having the minimum and maximum values, equal-width histograms can be very efficiently constructed in one pass through the values. Equal-depth histograms can be efficiently constructed, whenever the values are already sorted (as, e.g., in certain indices like B^+ -trees), and the estimations based on equal-depth histograms are typically more precise than estimations based on equal-width histograms when the values are not well distributed. The estimations based on most-frequent-values histograms are more precise than estimations based on the other types of histograms whenever some values occur very often, and the number of occurrences of all the other values is not significant.

The equal-width and equal-depth histograms divide the whole range of the input data into disjoint intervals. We denote $[l, u]$ for an interval for which $v \in [l, u]$ holds if $l \leq v \leq u$, and similarly, $v \in [l, u[$ holds if $l \leq v < u$. Let v_{min} be the minimum and v_{max} be the maximum value of the values to be represented by a histogram, then a histogram typically contains intervals $[v_{min}, u_0], [u_0, u_1], \dots, [u_{n-2}, u_{n-1}[$, $[u_{n-1}, v_{max}]$ with $v_{min} < u_0 < u_1 < \dots < u_{n-2} < u_{n-1} < v_{max}$ to cover the whole range from v_{min} to v_{max} . Due to simplicity of presentation, we focus on the interval type $[u_i, u_{i+1}[$ here, as the last interval $[u_{n-1}, v_{max}]$ can be also replaced with $[u_{n-1}, u_n[$, where u_n is a value with $v_{max} < u_n$. Note that the formulas presented for $[u_i, u_{i+1}[$ intervals can be slightly altered to deal with $[u_i, u_{i+1}]$ intervals. For each interval, usually the number of values and the number of distinct values, which are located at this interval, are stored in the histogram. Thus, an equal-width or equal-depth histogram H can be represented by a set of histogram entries consisting of the intervals $[l_i, u_i[$ with $l_i < u_i$, the number nv_i of values, and the number dv_i of distinct values in this interval (and therefore obviously $nv_i \geq dv_i$ must hold); that is, $H = \{([l_0, u_0[, nv_0, dv_0), \dots, ([l_n, u_n[, nv_n, dv_n)\}$, where $u_i \leq l_{i+1}$ for $i \in \{0, \dots, n-1\}$. According to this definition, we also allow “holes” in the intervals; that is, a histogram $\{([0, 5[, 5, 4), ([5, 7[, 3, 3), ([9, 15[, 9, 3)\}$ is valid, although $[7, 9[$ is not covered by the intervals of this histogram. The more intervals a histogram contains, the better the estimations are, but the longer it takes to compute the estimations. We therefore have a trade-off between preciseness and computation costs.

The equal-depth histogram chooses the intervals in such a way that every interval contains the same (or at least a similar) number of values. This type of interval can be efficiently constructed from a sorted list of values. Let p be the number of values of each interval and we assume k to be the number of intervals such that $k \cdot p$ is the number of values or a slightly larger number. Due to simplicity of presentation, we assume $k \cdot p$ to be exactly the number of values. Furthermore, let v_i be the i th value in the sorted list of values $v_1, \dots, v_{p \cdot k}$. The idea is that the first p values in the sorted list of values belong to the first interval, the values from $(p+1)$ to $2 \cdot p$ to the second interval, the ones from $(2 \cdot p+1)$ to $3 \cdot p$ to the third interval, and so on, such that the intervals $[v_1, v_{p+1}[$, $[v_{p+1}, v_{2 \cdot p}[$, \dots , $[v_{(k-1) \cdot p+1}, v_{k \cdot p}[$ are constructed. As the values $v_{i \cdot p}$ and $v_{i \cdot p+1}$ for $i \in \{1, \dots, k-1\}$ may be the same values and we construct the intervals $[v_{(i-1) \cdot p+1}, v_{i \cdot p+1}[$ and $[v_{i \cdot p+1}, v_{(i+1) \cdot p}[$, the numbers of values belonging to the intervals may be slightly different. This is also the case in our example in Fig. 5.5b), where the histograms consist of three intervals.

The equal-width histogram (Fig. 5.5a) divides the whole range of the input data into intervals with equal width. Let v_0 be the lowest value of the input data, w be the width of the intervals, then in an equal-width histogram the first interval has the borders $[v_0, v_0+w[$, the second interval $[v_0+w, v_0+2 \cdot w[$, the third interval $[v_0+2 \cdot w, v_0+3 \cdot w[$, and so on.

The most-frequent-values histogram (Fig. 5.5c) lists the most common values of the input data and their number of occurrences. If the most-frequent-values histogram does not come along with an equal-width or equal-depth histogram, then all the values different from the already considered most common values may be grouped and their number of occurrences and their number of distinct values stored. The most-frequent-values histogram can be therefore represented as $\{(a_0, n_0), \dots, (a_i, n_i), ([v_{min}, u_n[- \{a_0, \dots, a_i\}, nv, dv)\}$, where a_0, \dots, a_i are the most frequent values and n_0, \dots, n_i their numbers of occurrences, and $[v_{min}, u_n[- \{a_0, \dots, a_i\}$ is the whole range of considered values without the most frequent values containing nv values and dv distinct values.

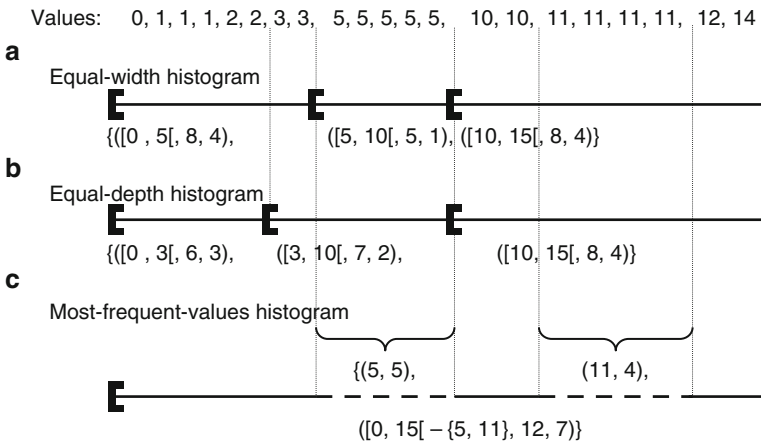


Fig. 5.5 Different types of histograms

Histograms can be easily built from the result of triple patterns. For all other operators in the logical operator graph, formulas to compute a resultant histogram from the histograms of previous operators exist in the scientific literature. However, these formulas are not based on exact science and are rather determined from experiences with experiments (see e.g., Garcia-Molina et al. 2002; Connolly and Begg 2002; Elmasri and Navathe 2000). We therefore do not present them here.

5.3 Further Related Work

Besides the specification of SPARQL (Prud'hommeaux and Seaborne 2008), several other works define the semantic of SPARQL as a part of their contributions; for example, Groppe et al. (2007b) describe a SPARQL evaluator for RDF data streams, (Groppe et al. 2008; Polleres 2007) describe the transformation from SPARQL queries to XQuery/XSLT and rules, and (Pérez et al. 2006) investigate the complexity of SPARQL evaluations.

The complexity of the evaluation of SPARQL queries is PSPACE-complete (Pérez et al. 2006, 2009), even if SPARQL does not contain any *FILTER* clauses. The evaluation complexity of some SPARQL fragments has better runtime complexities. For example, the graph patterns with only *AND* and *FILTER* operators are in $O(|P|*|D|)$, where $|P|$ is the size of the graph pattern and $|D|$ is the size of the data. The SPARQL fragment, which supports *AND*, *UNION*, and *FILTER* operators, is NP-complete.

Angles and Gutierrez (2008) show that SPARQL has the same expressive power as relational algebra.

Gutierrez et al. (2004) describe the semantics and complexity of conjunctive RDF query languages with triple patterns, which is the core concept of SPARQL. Serfiotis et al. (2005) describe some algorithms for a containment tester and the minimization of RDF/S query patterns, but they only consider hierarchy of concepts and RDF properties. Cyganiak (2005) and Frasnar et al. (2004) propose an algebra for SPARQL, which are derived from the relational algebra and which enable logical transformations for SPARQL queries in order to optimize the evaluation time. Klug (1988) describes an algorithm for the problem of containment testing of conjunctive queries containing inequalities. The algorithm is in Π_2^P , but it is open if the containment problem itself is in NP.

5.4 Summary and Conclusions

In this work, we define an algebra for SPARQL queries and the semantics of its processing, and we have discussed and developed equivalency rules based on this algebra. The equivalency rules are the basis for query optimization, where the task

is to choose a plan among possible equivalent ones with the best estimated costs. We described heuristic as well as query optimization approaches, which enumerate all possible query plans, estimate their costs, and choose the one with the best estimated costs. The main factor in the costs of a plan is the sum of the cardinality of its intermediate results. Therefore, for more systematic approaches to query optimization, we introduce histograms for cardinality estimations.

Chapter 6

Physical Optimization

Abstract Different algorithms exist to compute the result of a logical operator like AND, OPT, or SORT. A *physical operator* implements one of the algorithms to compute the result of a logical operator. The different physical operators sometimes have different constraints on the input data like that the input data must be sorted, or are faster than others for special types of input data, for example, when the input data fit into main memory. The context of an operator can be described by the estimations of properties of its input data. For each (logical) operator in the operatorgraph, *physical optimization* aims to choose the physical operator with the best estimated execution times in the operator's context.

As well as describing the physical operators, we in this chapter present our new approaches to efficient RDF data management and join optimization for small datasets and for large-scale datasets with over one billion triples.

For small datasets, where the data can be indexed in main memory, in-memory indices can significantly speed up query processing because (after loading the data) no disk accesses need to be done for query processing. B⁺-trees are optimized for disk indices of large-scale datasets, as they are optimized for blockwise sequential accesses of disks. For main-memory indices, hash indices are preferable as an index access can be done in constant time, as only a hash function must be applied to the key to retrieve the (main memory) address of the indexed element. Therefore, we use hash indices to manage small RDF datasets. Based on the triple nature of RDF data, we create seven hash indices in order to retrieve in-memory RDF data quickly. On the basis of the SPARQL-specific properties and the seven indices, we develop a new, efficient approach to computing join by dynamically restricting triple patterns. A performance evaluation demonstrates that the new approach outperforms other state-of-the-art in-memory databases.

Since the Semantic Web datasets are becoming increasingly large, developing efficient techniques to speeding up querying large-scale Semantic Web data is a key issue for Semantic Web applications. When data are already sorted, from relational database research, merge joins are known to be the fastest join algorithms on large-scale data. Therefore, recent approaches focus on the presorting of Semantic Web data during index construction, and thus the fast merge join can be used without a sorting phase at runtime for some joins. When data for succeeding joins become unsorted, the hash join is typically used. In this chapter, we propose a sorting

numbering scheme for large RDF datasets, based on which we can fast sort any intermediate and final querying results. Applying our sorting numbering scheme, all joins can be computed using the merge join with a fast sorting phase. Besides being a significant benefit to merge joins, our fast sorting technique can also remarkably speed up the elimination of duplicates. Our experiments show that a merge join using our fast sorting technique outperforms greatly the hash join and that our sorting numbering scheme integrated into any index approaches significantly speeds up querying large-scale Semantic Web data.

6.1 Motivation

Semantic Web ontologies and RDF knowledge bases are becoming increasingly large. The examples of large RDF data with millions and even billions of facts include the UniProt comprehensive catalogue of protein sequence, function, and annotation data (Swiss Institute of Bioinformatics 2009), the RDF data extracted from Wikipedia (Auer et al. 2007), the Princeton University’s WordNet (Assem et al. 2006), and the Billion Triples Challenge (Semantic web challenge 2009). Examples of other RDF data include RSS 1.0 (Beged-Dove et al. 2001) and FOAF (Brickley and Miller 2007). Therefore, an important research task is developing efficient approaches to processing SPARQL queries over very large RDF data.

An amount of work (e.g., Chong et al. 2005; Guha 2010; Harris and Gibbins 2003; Pan and Heflin 2003; Volz et al. 2003; Wilkinson 2006) maps the RDF data format to the format of relational databases and SPARQL queries to SQL queries, thus leveraging the proved database technologies. However, the relational optimizations do not specialize on the data model of RDF triples and the usage of SPARQL triple patterns. Furthermore, this kind of approach also fails to handle very large RDF databases (see Abadi et al. 2007; Neumann and Weikum 2008, 2009; Weiss et al. 2008).

As well as the common and similar properties between the RDF data and the relational tables, and between SPARQL and SQL, RDF and SPARQL also have their own properties. Relational optimization techniques do not specialize on the data model of triples and the usage of triple patterns in query languages. These RDF- and SPARQL-specific features have attracted the attention and interests of researchers to develop new optimization techniques, for example, Abadi et al. (2007) and its generalization in Weiss et al. (2008) and Neumann et al. (2008). Weiss et al. (2008) and Neumann and Weikum (2008) use six indices according to the six possibilities SPO, SOP, PSO, POS, OSP, and OPS to order RDF triples. For example, the SPO collation order regards the subject (S) of an RDF triple as primary order criterion, the predicate (P) as secondary, and the object (O) as tertiary order criterion. However, (Weiss et al. 2008; Neumann and Weikum 2008) still apply conventional relational merge join algorithms to compute the joins of triple patterns. These contributions do not fully exploit RDF- and SPARQL-specific properties for optimizations of, for example, in-memory join computation, which

are also studied in this work. The approaches (Abadi et al. 2007; Neumann and Weikum 2008, 2009; Weiss et al. 2008) avoid costly self-joins in one large triple table and can handle quite large-scale Semantic Web data. For the first several joins, the fast merge joins can be directly applied on already sorted data. However, in general, not all joins can be computed with the merge join algorithm without requiring extra sorting phases at runtime. This happens already for nonbushy queries with only three triple patterns, for example, $(?a < origin > < DLC > .)$, $(?a < records > ?c .)$ and $(?c < type > ?b .)$. When data become unsorted for succeeding joins, hash joins are used in these approaches. From the relational database research, hash joins are known to be the fastest approach to computing joins, which do not require sorted input data. Hash joins are very efficient and cheap if at least one of two operands of the join can fit into main memory. However, when querying the very large Semantic Web databases, one cannot assume that the data for hash joins can always or even often fit into main memory.

In this chapter, we suggest a sorting numbering scheme for efficiently querying large-scale Semantic Web databases. On the basis of our sorting numbering scheme, we develop a fast sorting approach. When the data for succeeding joins become unsorted and are too large to fit into memory, using the merge joins with our fast sorting technique is more efficient than using a hash join on the unsorted data. Furthermore, our fast sorting technique is of great benefit to the queries, which require duplicate elimination. The sorting numbering scheme can be integrated into any index approaches like the ones of (Neumann and Weikum 2008, 2009; Weiss et al. 2008) to speed up querying the very large Semantic Web databases.

Overall, the contributions of our work to large-scale datasets include as follows:

- A sorting numbering scheme based on the RDF- and SPARQL-specific properties for managing and efficiently querying large-scale Semantic Web databases
- A fast sorting technique based on the sorting numbering scheme for computing any joins using the fast merge join approach, and for eliminating duplicates efficiently
- Integration of our sorting numbering scheme into index approaches for speeding up querying very large Semantic Web data
- The idea of using integer identifiers of RDF terms as presorting numbers for fast sorting intermediate and final results of queries, such that our approach of sorting numbering can efficiently support updates and does not need any additional storage space
- A concept-proof prototype, including the implementations of our sorting numbering schemes and fast sorting algorithms, reimplementations of several existing approaches, and integration of our approach into these existing approaches, in order to compare different approaches
- A performance analysis, which demonstrates that the application of our fast sorting technique significantly speeds up the join computation and duplicate elimination when querying large-scale Semantic Web databases

Besides increasingly larger datasets, the main memory sizes of typical computer configurations increase continually. Therefore, more and more datasets used in real-world applications can be managed completely in main memory, and thus also in-memory databases become increasingly important. Just applying optimizations for large-scale datasets in in-memory databases lead to suboptimal query processing, that is, special optimizations using the elements in main memory that can be directly addressed away from constraints of sequential disk accesses can boost query evaluation. Our approach for in-memory databases joins triple patterns by dynamically restricting triple patterns, that is, joining one triple pattern to the solution of the previous triple pattern or of the join of previous triple patterns. In order to compute the join of two triple patterns, we first compute one triple pattern and then use the resultant data to replace the corresponding variables in another triple pattern. In this way, the join computation only involves retrieving from the given RDF data. Therefore, we only need to create seven hash indices (S, SP, SPO, SO, P, PO, and O using keys on the subject (S), predicate (P), and/or object (O)) on the original RDF data in order to fast retrieve data specified by any triple pattern. In comparison with the indices described in (Weiss et al. 2008; Neumann and Weikum 2008), where accessing the index requires time-consuming searches in B^+ -trees and sorted lists, using our indices, we can access the result of any triple pattern with *one* index access in main memory.

The contributions of this chapter for in-memory databases include as follows:

- An approach to manage and access RDF data efficiently using seven indices
- An approach to efficiently compute joins for in-memory database engines
- An experimental evaluation, which shows that our approach is faster than in-memory adaptations of disk-based approaches (e.g., Weiss et al. 2008) and other existing state-of-the-art in-memory engines for SPARQL processing

6.2 Related Work

Several index approaches are developed to manage RDF data for efficient query processing. Abadi et al. (2007) suggest a vertical partitioning approach to storing RDF data. In this scheme, the RDF triples are stored in two-column tables, and each table manages one property and contains a subject and an object column. Each table is sorted by subject, and thus particular subjects can be accessed quickly, such that fast merge joins can be applied while joins are processed on the subject. Abadi et al. (2007) employ a column-oriented DBMS (e.g., Stonebraker et al. 2005) to manage these property tables in order to leverage its benefits of compressibility and performance. The property tables in (Abadi et al. 2007) have considerable advantages for the SPARQL triple patterns, where the predicate is a RDF term but not a variable. However, this approach does not sufficiently support the efficient processing of the
(continued)

triple patterns, where the predicate is a variable (see [Neumann and Weikum 2008, 2009](#); [Weiss et al. 2008](#)).

For efficient processing of more general queries, *Hexastore* ([Weiss et al. 2008](#)) and *RDF3X* ([Neumann and Weikum 2008, 2009](#)) use six indices corresponding to the six collation orders SPO, SOP, PSO, POS, OSP, and OPS to manage RDF triples. Depending on which positions in a triple pattern contain RDF terms (e.g., the subject and the object), one of the indices (e.g., SOP) is used to efficiently retrieve the data by using a prefix search. Using these collation orders, some joins can be computed using the fast merge join approach over sorted data. In comparison, our approach optimizes the evaluation of remaining joins, when data become unsorted.

For every collation order, for example, SPO, ([Weiss et al. 2008](#)) proposes to associate a subject key s_j to a sorted vector of n_i property keys, $\{p_{1j}^i, p_{2j}^i, \dots, p_{n_i j}^i\}$. Each property key p_{ij}^i is, in its turn, linked to an associated sorted list of $k_{i,j}$ object keys. These object lists are shared in indices for corresponding collation orders; for example, the object lists for SPO are also shared by the index for PSO. *RDF3X* ([Neumann and Weikum 2008, 2009](#)) uses a more elegant solution to store data sorted according to the six collation orders: employing just B^+ -trees and prefix searches and thus gaining a simpler and faster index structure than ([Weiss et al. 2008](#)).

RDF3X ([Neumann and Weikum 2008, 2009](#)) and *Hexastore* ([Weiss et al. 2008](#)) use sophisticated data structures to compress their index structures. *RDF3X* ([Neumann and Weikum 2008, 2009](#)) also supports additional special aggregated indices for fast processing of a special kind of queries. However, important features of SPARQL like data types are currently not supported by *RDF3X*. [Neumann and Weikum \(2008, 2009\)](#) also describe some cardinality estimation techniques of SPARQL join results and a sophisticated plan generator. Furthermore, ([Neumann and Weikum 2009](#)) introduce the Sideways Information Passing (SIP) strategy for optimizing query processing.

If meta nodes are ignored, the approaches described in ([Harth and Decker 2005](#)) and in ([Groppe et al. 2009a](#)) use the seven indices S, SP, SPO, SO, P, PO, and O for retrieving the result of a triple pattern within one index access. When B^+ -trees and prefix searches are used, then the number of indices can be reduced to four (accesses to the S and SP indices can be answered by a prefix search in the SPO index; accesses to the P index by the PO index). While ([Harth and Decker 2005](#)) show that four B^+ -tree indices work well for disk-based Semantic Web applications, ([Groppe et al. 2009a](#)) demonstrate that seven hash indices perform well for in-memory SPARQL engines. ([Groppe et al. 2009a](#)) first compute one triple pattern and then use the resultant values to replace the corresponding variables in the following triple patterns. In this way, the join computation only involves the retrieving from the given RDF data, and thus the 7 indices on the original data are enough, and neither sorting nor hashing of the solutions are needed. However, this join

(continued)

approach becomes inefficient when the data do not fit into memory any more because it causes extra disk accesses.

The SPARQL-engine Kowari (Wood et al. 2005) envisions statement-based queries. A statement-based query lacks one or two parts of a triple, and its answer is a set of resources that complement the missing parts. If meta nodes are ignored, the number of required indices of the Kowari solution is 3, defined by the three cyclic orderings SPO, POS, and OSP. Since the other three indices SOP, PSO, and OPS are missing, Kowari cannot efficiently process general queries.

Other SPARQL engines such as Jena (see Wilkinson 2006), 3store (Harris and Gibbins 2003), DLDB (Pan and Heflin 2003), KAON (Volz et al. 2003), Oracle (Chong et al. 2005), and rdfDB (Guha 2010) utilize a traditional relational database or Berkeley DB as their underlying persistent data store (Matono et al. 2005). Most of these SPARQL engines store RDF triples directly in relational or hash tables, and thus simple statement-based queries can be satisfactorily processed by such systems. However, the conventional approaches are not efficient for more complex queries (Matono et al. 2005) involving, for example, multiple filtering steps.

Some systems such as Jena (Wilkinson 2006) attempt to create relational-like property tables out of RDF data, and these tables gather together information about multiple properties over a list of subjects. Still, these schemes do not perform well for queries that need to combine data from several tables (Abadi et al. 2007). A relational-like structure on RDF data results in a sparse representation with many NULL values in the formed property tables. Handling such sparse tables, as opposed to denser ones, requires a significant computational overhead (Abadi et al. 2007).

Angles and Gutiérrez (2005) and Hayes and Gutiérrez (2004) deal with the possibility of storing RDF data as a graph, but do not sufficiently address the scalability questions either. Matono et al. (2003) and Kim et al. (2005) propose a path-based approach for managing RDF data and store subgraphs into distinct relational tables. These systems do not provide the scalability necessary for querying large-scale data. As well as (Neumann and Weikum 2008, 2009; Bernstein et al. 2007) also use selectivity estimation techniques for query optimization. However, this approach focuses on small RDF graphs, which fit into main memory, and thus also faces scaling problems. Liarou et al. (2007) investigate SPARQL extensions for handling continuous queries.

6.3 Indexing

An index stores key-value pairs in such a datastructure that a value can be efficiently retrieved for a given key. Query processing is often speeded up by storing the input data in indices and by accessing the results of query subexpressions using these

indices. In the following subsections, we describe in-memory and disk-based indices for the processing of SPARQL queries.

6.3.1 Building In-Memory Indices

In this section, we first focus on in-memory indices, where the input data fits into main memory. In main memory, spread elements can be addressed directly via, for example, hash tables and we do not need to consider *sequential* disk-based accesses. We describe the usage of seven indices in order to retrieve the result of any triple pattern with one index access. RDF is a set of triples $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$. Triple patterns of SPARQL queries contain either constant values or variables, for example, $(?article \text{ rdf:type } \text{bench:Article})$. In order to access the result of any triple pattern $(s \ p \ o)$ efficiently, we use the sequence of constant values in triple patterns as keys to create indices. Therefore, for any RDF data and SPARQL queries, we only need to construct seven indices using $s, p, o, sp, so, po,$ and spo as keys, respectively. The seven indices are enough for any triple pattern in SPARQL queries to retrieve RDF data quickly. As our experiments show, constructing and managing seven indices is also practical even for large RDF data, which still fit into main memory.

Example 1 (Index with sp as key). The following RDF graph is the input RDF graph:

$$D = \{ \begin{array}{l} (:article1, \text{rdf:type}, \text{bench:Article}), \\ (:article1, \text{rdf:type}, \text{bench:JournalArticle}), \\ (:article1, \text{dc:creator}, :person1), \\ (:inproc1, \text{rdf:type}, \text{bench:Inproceedings}) \end{array} \}$$

The following table presents the index with sp as key and the triples for the corresponding key:

Key sp	Triples
$:article1 \text{ rdf:type}$	$\{ (:article1, \text{rdf:type}, \text{bench:Article}), \\ (:article1, \text{rdf:type}, \text{bench:JournalArticle}) \}$
$:article1 \text{ dc:creator}$	$\{ (:article1, \text{dc:creator}, :person1) \}$
$:inproc1 \text{ rdf:type}$	$\{ (:inproc1, \text{rdf:type}, \text{bench:Inproceedings}) \}$

If we use indices, which allow a prefix search like supported by B^+ -trees but not by hash tables, then we can reduce the number of indices to four, as a s -index access and a sp -index access can be answered by a prefix search on the spo index, and a p -index access by a prefix search on the po index. While B^+ -trees or variants of B^+ -trees seem to be the best choice for indices of disk-based approaches (see Weiss et al. 2008; Neumann and Weikum 2008), our experimental evaluation shows that hash tables are the best choice for in-memory databases, as one index access can be done in constant time.

The seven indices are generated only once when reading the input data and can be afterward used for the evaluation of any SPARQL query during the lifetime of

the in-memory database engine process. As we will see later on, for joining the results of any triple patterns, we need only these seven indices.

6.3.2 Building Disk-Based Indices

Most Semantic Web query evaluators use B^+ -trees to store large-scale RDF datasets on disks (e.g., Weiss et al. 2008; Neumann and Weikum 2008, 2009). B^+ -trees can be built very efficiently from a sorted list of data by avoiding expensive node splitting. Among a number of sorting algorithms, *merge sort* scales well for very large data and performs especially well for external sorting. Therefore, our SPARQL engine uses the merge sort technique to sort data for constructing indices efficiently. In the merge sort algorithm, our SPARQL engine uses a sort&merge-heap (Groppe and Groppe 2010) with replacement selection (Friend 1956) in order to increase the size of the initial runs. We now describe the internals of our SPARQL engine for large-scale datasets. Note that the internals of other SPARQL engines may slightly differ, but the main principles are the same. Three different types of indices are created and maintained in our SPARQL engine: dictionary indices, evaluation indices, and histogram indices.

6.3.2.1 Dictionary Indices

Semantic Web query evaluators such as RDF3X (Neumann and Weikum 2008, 2009) and Hexastore (Weiss et al. 2008) as well as our SPARQL engine use dictionary indices to map RDF terms into integer ids. One advantage of ids is lower space requirements in the evaluation indices storing the input RDF triples as an integer is stored instead of a possibly large string. Furthermore, more space can be saved by avoiding storing leading zero bytes and using difference encoding, which we will explain in detail in the section about the evaluation indices. Solutions using ids consume less space such that the memory footprint is smaller and/or more solutions can be processed without swapping to hard disks and thus improving the performance. Using ids have disadvantages in seldom cases when operations like sorting or relational comparisons like $<$, $<=$, $>=$, and $>$ require the RDF terms instead of the ids causing high costs for large intermediate results because of the materializations of the RDF terms. Furthermore, displaying the final query result has also high costs whenever the query result is large. However, the advantages typically outweigh the disadvantages of using ids for large-scale datasets.

One dictionary index maps RDF terms into integer ids; one translates integer ids back into RDF terms. The dictionary indices do not fit into main memory for large-scale datasets such that our SPARQL engine uses B^+ -trees for the dictionary indices. When storing RDF terms in the dictionaries, we use difference encoding in order to save storage space: we determine common left substrings of the current

and previously stored strings and store only the length of the common left substring together with the remaining right substring of the current string. Furthermore, after transforming id values of query results back to RDF terms, we cache the RDF terms with their ids together in order to avoid multiple materializations. We use the strategy of least recently used (LRU) caches for the accesses to the B⁺-tree nodes in order to further improve the performance of these materializations.

The dictionary indices are used to transform RDF triples into id triples, which are consisting of ids instead of RDF terms and are then stored in the evaluation indices: Id triples are obtained from RDF triples by using the dictionary index from strings to ids and mapping the RDF terms of the subject, predicate, and object from the triples to their ids. If many RDF triples must be transformed into id triples like when importing a large dataset into the database, then it is not efficient to query the dictionary index for every single RDF term accessing a path from the root to a leaf of the B⁺-tree for mapping RDF terms into ids. It is more efficient to use three passes through the RDF triples, where first the subjects, then the predicates, and finally the objects are transformed into ids. In each pass, the triples are first sorted according to the component (subject, predicate, or object) to be transformed into ids. Afterward, we iterate through the sorted RDF triples and simultaneously through the sorted RDF terms of the dictionary index. We read RDF terms from the dictionary index until we have found the entry for the current component of the RDF triple and replace this component with the corresponding id. Afterward, we read the next RDF triple and proceed as before. In this way, we only need one pass through the sorted RDF triples to be imported and the dictionary index to transform one component of the RDF triples into their ids. Furthermore, we can also use SIP strategies as we have discussed when introducing B⁺-trees to increase the performance during iterating through the dictionary index. Afterward, we sort our (partly) transformed triples to be imported according to the next component and replace their RDF terms with the corresponding ids like before until all components are transformed into ids.

6.3.2.2 Evaluation Indices

These indices are used for the evaluation of SPARQL queries. They are constructed from sorted id triples according to the six collation orders SPO, SOP, PSO, POS, OSP, and OPS of RDF. Large-scale datasets are hard to manage without compression. Using (integer) ids instead of RDF terms already compresses the indices much. We can compress such indices further by storing only the different components of a triple in comparison to the last stored triple. For example, assuming the collation order SPO and a last stored triple ($\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$), we only need to store the object $\langle d \rangle$ for a triple ($\langle a \rangle$, $\langle b \rangle$, $\langle d \rangle$). Of course, we have to additionally store two bits for distinguishing if all three components of a triple must be stored, two components, or only one component. An integer id of an RDF term is typically represented by four bytes. For small integer ids, some leading bytes of these four bytes are zero. We can now store two bits for distinguishing if no, one,

two, or three leading bytes are zero, and only store the bytes without leading zeros. We further know that the id $i1$ of the triple's object must be larger than the id $i2$ of the last triple's object in our example because of the SPO collation order. Therefore, we only need to store the difference $i1-i2$, which may have more leading zero bytes. However, for the object of triples, where numerical values and language-tagged literals can occur, we must consider special RDF properties, which we explain in the following paragraph:

In RDF data, typed literals like integer values of XML Schema can have several representations, for example, 2 and +2. During query processing, they are treated as value-equal integer values. Therefore, they should be assigned with the same id. Otherwise, some processing, like the computation of join, might create wrong results. However, the W3C test cases (Feigenbaum 2008) show that the query results must contain the original representation. Similar remarks also apply to language-tagged literals. For example, "Text"@DE and "Text"@de are treated as equal values, but the original representation must be maintained for the final result. Therefore, we additionally store an id, which refers to the original representation in the index, additionally to the id for value-equal literals, if necessary. Different representations of equal values are only possible for typed literals and language-tagged literals, which can only occur in objects of triples. Therefore, we only need to store an additional id for the objects if the original representation differs from the indexed one. In comparison, the original RDF3X prototype (Neumann and Weikum 2008, 2009) does not support data types, and considers, for example, the same integer values 2 and +2 to be two different literals.

Furthermore, every node in our B⁺-trees has an integer id, which allows us to highly compress references to B⁺-tree nodes as well. Figures 6.1 and 6.2 present the stored bits and bytes for a leaf in a B⁺-tree, when an id triple is stored (see Fig. 6.1) or a reference to the next B⁺-tree node (see Fig. 6.2). Figure 6.3 presents the stored bits and bytes for a B⁺-tree interior node, when a key in form of an id triple with a reference to its B⁺-tree child node is stored. The last entry of a B⁺-tree interior node contains only a reference to a B⁺-tree child node. For this last entry, we use the same stored bits and bytes as presented in Fig. 6.2 for a B⁺-tree leaf: We just store the reference to the B⁺-tree child node instead of the reference to the next B⁺-tree leaf.

The original RDF3X prototype (Neumann and Weikum 2008, 2009) supports additional special aggregated indices for fast processing a special kind of queries. For example, the two triples ($\langle a \rangle$, $\langle b \rangle$, $\langle C \rangle$) and ($\langle a \rangle$, $\langle b \rangle$, $\langle D \rangle$) are aggregated as ($\langle a \rangle$, $\langle b \rangle$, 2) in the aggregated SP index in order to represent two triples with a common subject $\langle a \rangle$ and a common predicate $\langle b \rangle$ (but with different objects). The SP index can be used to fast retrieve the result of a triple pattern $\langle a \rangle ?p ?o$, if the variable $?o$ is neither further used nor occurs in the final result. Considering that such cases occur seldom in real world and the additional costs for maintaining the aggregated indices, aggregated indices are not supported by our SPARQL engines.

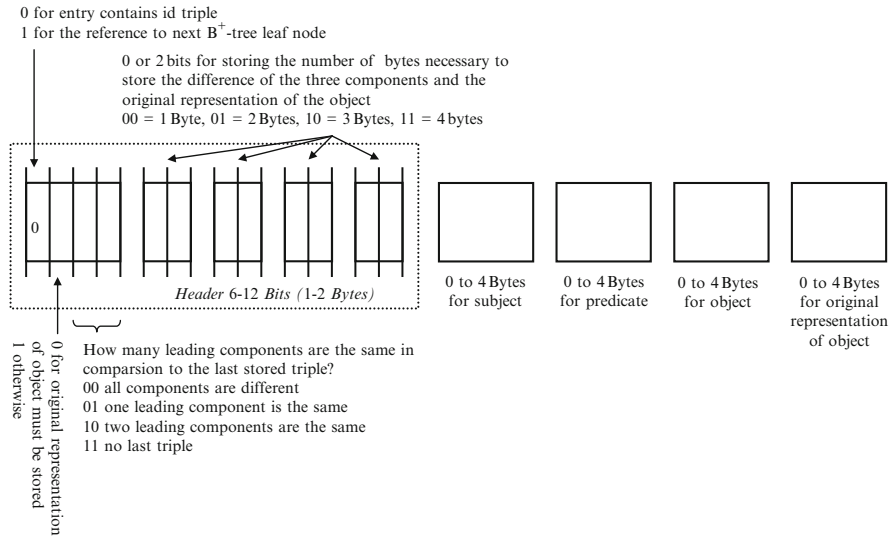
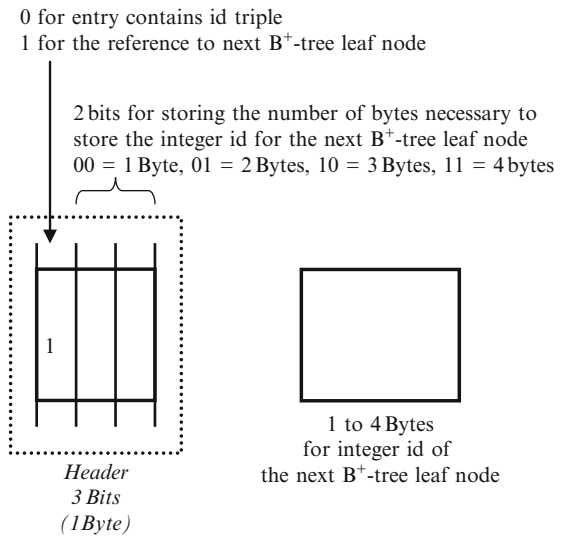


Fig. 6.1 Entry for id triple in B⁺-tree leaf

Fig. 6.2 Entry for integer id for next B⁺-tree leaf node



6.3.2.3 Histogram Indices

Our plan generator uses equi-depth histograms (Piatetsky-Shapiro and Connell 1984) for estimating the cardinality of results of triple patterns and for calculating the overall cost of a plan. A histogram is created for a triple pattern and a specific

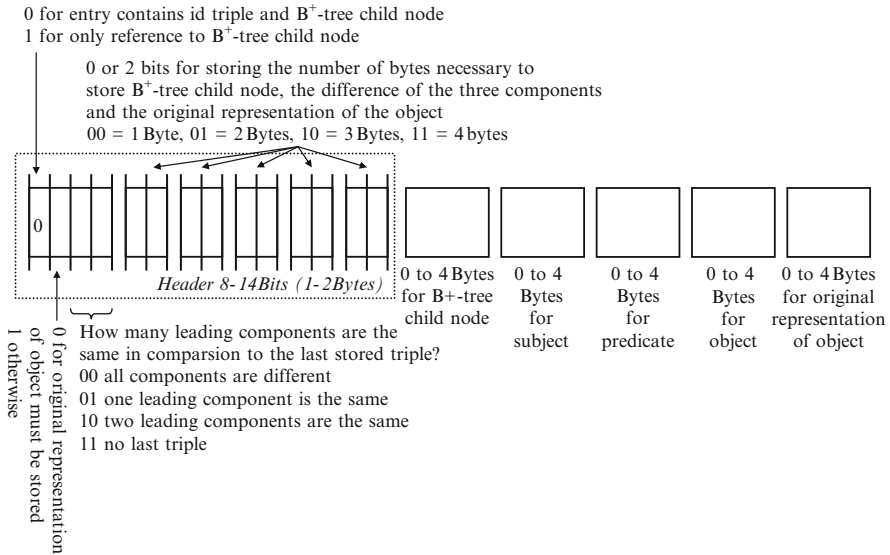


Fig. 6.3 Entry for id triple and B⁺-tree child node in B⁺-tree inner node

variable of it. Each interval in the histogram contains the number of the triples allocated in this interval and the numbers of distinct values. In order to speed up the generation of equi-depth histograms, we use a special B⁺-tree for each collation order. In each inner node of this special B⁺-tree, we store the number of triples, the number of distinct subjects, predicates, and objects, and three bits. The three bits indicate whether the subject, predicate, or object of the first triple *F* in the subtree is different from the triple before *F*.

Using this special B⁺-tree and especially its additional information B, we can very quickly find the corresponding information related to a triple pattern by not only passing leaf nodes, but also jumping over whole subtrees. Therefore, histograms of triple patterns can be constructed very efficiently from these histogram indices. This is also shown by our experimental results. Figure 6.4 illustrates how to construct the histogram for the variable ?v and the triple pattern (3, ?v, ?o) from a histogram index. Since these additional B⁺-trees are only needed for efficiently computing histograms, their updates can be delayed to the times with low workload.

Once a histogram has been calculated, it is stored in a separate index and can be reused for the triple patterns with the same RDF terms and variables at the same positions, but independent from the names of variables. While histogram computations using histogram indices need some seconds for large-scale datasets like the one from the Billion Triples Challenge (Semantic web challenge 2009), reusing a histogram by retrieving it from the separate index only consumes milliseconds.

Other contributions to Semantic Web databases do not use histogram indices: (Weiss et al. 2008) do not use any histograms for query optimization, which leads to inefficient query plans. Neumann and Weikum (2008) compute histograms using

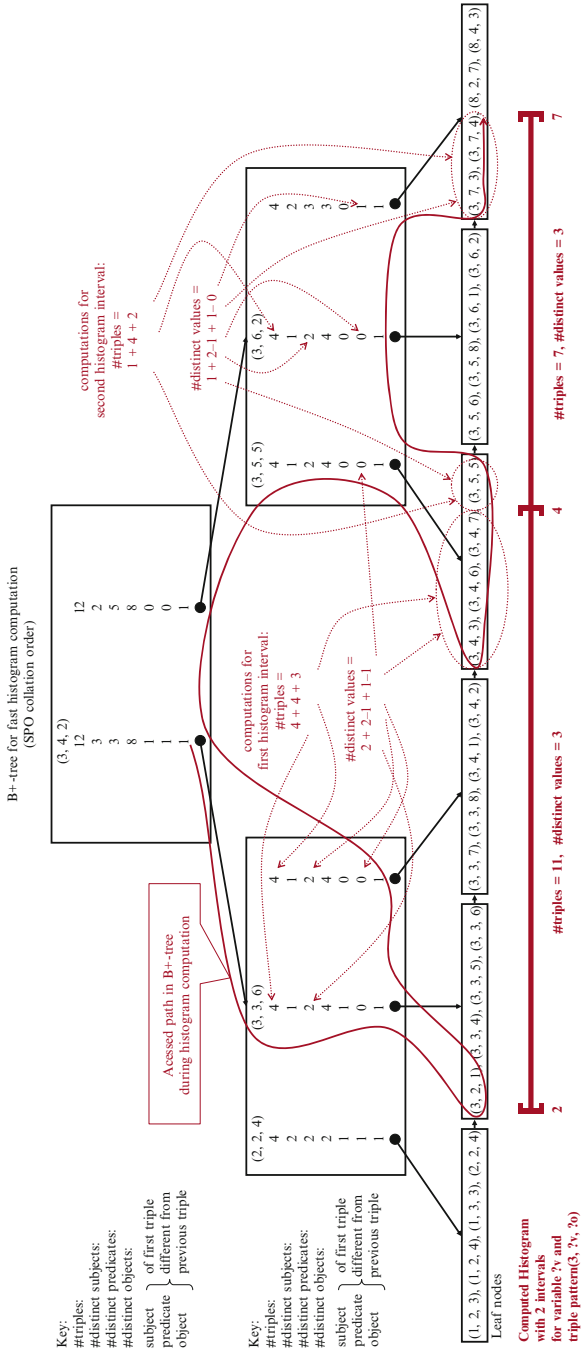


Fig. 6.4 Fast construction of the histogram for the variable ?v and the triple pattern (3, ?v, ?o) using a histogram index

the evaluation indices, which is too slow for large datasets. Neumann and Weikum (2009) precompute all possible joins facing problems with efficiency and cost estimations behind joining only two triple patterns.

6.4 Pipelining Versus Materialization

Many physical operators can determine first solutions after some and not all input data come in and can deliver these solutions early to its succeeding operators, which is called *pipelining*. For the support of pipelining, an operator has to support the iterator concept. The iterator concept requires the support of the methods *open()* for starting the computation, *next()* to retrieve the first (just after an *open()* call) as well as the next result, and *close()* to end computing and to free resources. The *open()* method of an operator calls the *open()* methods of its operands. The *next()* method gets solutions of the operator's operands by calling their *next()* methods until a solution can be determined, which is then returned. Thus, *next()* returns the solutions of an operator one by one. The *close()* method calls the *close()* methods of its operands before freeing its own resources.

The opposite *materialization strategy* first finishes the computation of an operator and determines the whole result and then proceeds to the next one. Few solutions may fit into main memory, but a large number of solutions must be stored on disk. Consequently, the costs for disk accesses slow down the performance. Contrary to the materialization strategy, pipelining does not require any materialization of intermediate results, as the solutions of operator results are computed one by one on demand of the parent's operator when calling the *next()* method avoiding huge costs for disk accesses.

6.4.1 Pipeline-Breaker

Some operators like the Sort operator cannot start the computation until all the data come in. These operators are called *pipeline-breaker*. Nevertheless, these operators can still support the iterator concept by just reading all the input data after an *open()* call, computing its result, and returning them one by one after *next()* calls. Thus, an operator does not need to know if its parent's operator is a pipeline-breaker or not and can just use the iterators for accessing its input.

6.4.2 Sideways Information Passing

SIP passes information from one operand to the other sideways in the operator-graph. For example, a merge join algorithm requires the input of its operands to be

sorted. Furthermore, if a solution A is read from one of the operands, then solutions of the other operand are read by the merge join algorithm until a solution equal to or larger than A is read. In a more intelligent way, we can pass A to the other operand and the other operand may optimize retrieving a solution equal to or larger than A . For this purpose, we extend the iterator concept for SIP, such that a *next(lowerLimit)* method must be supported. *next(lowerLimit)* methods return a solution equal to or larger than *lowerLimit* for sorted results. Note that this method has been already introduced for prefix searches in B^+ -trees, such that index scans retrieving the result of triple patterns are already optimized for processing *next(lowerLimit)* method calls. Furthermore, the information *lowerLimit* may be passed not only between the operands of an operator, but also to their operands, such that the optimization potential is enormous.

6.5 Join Algorithms

We first review the traditional join algorithms Nested-Loop, Merge, Index, and Hash Join. Afterward, we present our new approaches to join computation for the in-memory and large-scale RDF databases.

6.5.1 Nested-Loop Join

The nested-loop join is one of the simplest join algorithms and performs well for very small unsorted data. Therefore, this join algorithm is often used as part of other more complex join algorithms like the hash join.

In its simplest form, the nested-loop join contains a nested loop iterating through the solutions of its left and right operand. In each iteration, if the two solutions of the left and right operands can be joined, the joined result is returned. Otherwise, the loops proceed. Let R and S contain the solutions of its left and right operands, *joinable*(r, s) be a function to check if r and s can be joined, and *join*(r, s) computes the join between r and s . The following pseudo code presents the nested-loop join algorithm:

```

FOR EACH  $s$  IN  $S$  DO
  FOR EACH  $r$  IN  $R$  DO
    IF (joinable( $r, s$ )) {
      OUTPUT join( $r, s$ );
    }

```

The runtime complexity is $O(|S|*|R|)$. If the results of both operands of a join do not have any variables in common, then every solution of one operand must be combined with every solution of the other operand; that is, the join degenerates to a Cartesian product. Therefore, the *worst case* runtime complexity is $O(|S|*|R|)$ for

any join algorithm. However, the *average* runtime complexity for typical input data is often better for other join algorithms up to linear complexity $O(|S| + |R|)$ for merge joins, which may be $O(\min(|S|, |R|))$ or even better when using SIP strategies.

6.5.1.1 Iterator Version

In this subsection, we describe the iterator version of the nested-loop join algorithm. We therefore present the pseudo code of the three methods *open()*, *next()*, and *close()* here as follows:

```

open() {
    R.open();
    S.open();
    s = S.next();
}

next() {
    DO {
        r = R.next();
        IF(r == null) { // R is exhausted for the current r
            R.close();
            s = S.next();
            IF(s == null) { // both R and S are exhausted!
                return null;
            }
            R.open();
            r = R.next();
        }
    } WHILE (!joinable(r, s));
    RETURN join(r, s);
}

close() {
    R.close();
    S.close();
}

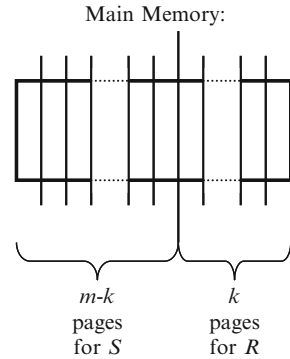
```

The new main ideas and principles of the other physical operators can be understood from the materialization versions of these physical operators. Therefore and due to simplicity of presentation, we will not present the iterator version for the other physical operators and discuss only the version for the materialization strategy.

6.5.1.2 Block-Based Nested-Loop Join

Database systems usually work on pages, which are arrays of bytes with fixed length (typically 8 kb). A page contains input data, metainformation, and solutions. Pages are stored on disk and loaded into main memory (using a buffer manager) for reading and manipulating. Solutions (and data) are typically organized in pages to

Fig. 6.5 Reserving pages for S and R in main memory



be controlled by the buffer manager. The buffer manager holds frequently accessed pages in main memory, such that the page(s) containing solutions (and data) may be never (temporarily) stored on disk if they fit completely into main memory.

The design of some physical operators pays attention to the page-oriented organization of data and optimizes the number of page accesses. These operators assume that a part of the main memory with the size for m pages is available for them.

The block-based nested-loop join reserves k pages for the result R of its first join operand and thus $m - k$ pages for the result S of its second join operand (see Fig. 6.5). According to the iterator concept, the solutions of the join are computed one by one after each call of *next()*. Thus, the join operator does not reserve any page for the result of the join. However, the operator calling *next()* of the join may reserve a page for the result of the join and may temporarily store the result on disk if necessary.

The block-based nested-loop join works as follows (see Fig. 6.6): First of all, it loads $m - k$ pages of S and k pages of R into main memory and then joins the $m - k$ pages of S with the k pages of R using, for example, the “normal” nested-loop join algorithm discussed in the previous subsections. Afterward, it loads the next k pages of R into main memory and joins these k pages of R with the already loaded $m - k$ pages of S . The process is repeated until all pages of R are joined with the first $m - k$ pages of S . Note that in the last round $\leq k$ pages are joined with the first $m - k$ pages of S . The next $m - k$ pages of S are then loaded into main memory and joined with R in the same way as the first $m - k$ pages of S . The process is repeated until all pages of R are joined one time with all pages of S , and therefore whole R is joined with whole S .

We have an exercise at the book webpage <http://www.ifis.uni-luebeck.de/~groppe/SemWebDBBook/> for analyzing the number of pages accessed by this variant of the nested-loop join.

After each round joining $m - k$ pages of S , k pages of R are still in main memory, which are then replaced with the first k pages of R to be joined with the next $m - k$ pages of S . An optimization of the presented block-based nested-loop

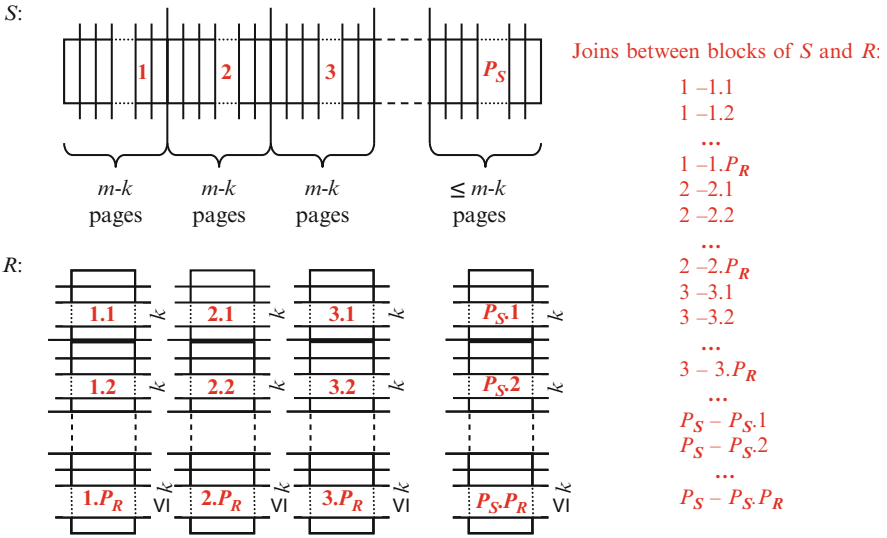


Fig. 6.6 Block-based nested-loop join

join uses these k pages of R from the last round and joins them with the next $m - k$ pages of S before this optimization proceeds joining with the first k pages of R . In this way, k pages are less loaded in each round (except of the initial round).

6.5.2 Merge Join

Let V_L be the set of bound variables of the left operand of a join, and V_R be the set of bound variables of the right operand. We call the variables in $V_L \cap V_R$ the *join variables*. The merge join requires the solutions of its operands sorted in the same way according to its join variables. We assume that the input is sorted in descendant order, such that the solutions of an operand are equal to or larger than the previous solutions. Note that the merge join algorithm can be easily adapted to consume sorted data in ascendant order by exchanging larger than comparisons in the pseudo code given below with smaller than comparisons. For an example of the merge join algorithm applied to the solutions of its operands, see Fig. 6.7. The merge join algorithm first reads the solutions from both operands and checks if they are joinable (i.e., the bound values of the join variables are the same). If they are *not* joinable, the next solution of the operand with the smaller values is read, because we are sure that the remaining solutions of the other operand are equal or larger values according to the sort criterion and thus cannot be joined with the smaller value. In the case that they are joinable, the merge join algorithm reads the next solutions from both operands until the bound values of the join variables differ from

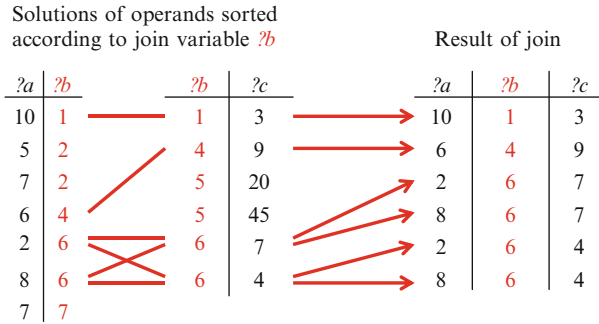


Fig. 6.7 Merge join example

the first read solution. All the read solutions with the same bound values of the join variables of both operands must now be joined and returned. The pseudo code for the materialization strategy variant of the merge join algorithm is therefore as follows:

```

S.open();
R.open();
s=S.next();
r=R.next();
WHILE (s!=null && r!=null){
  IF(s < r){
    s=S.next();
  } ELSE IF(r < s){
    r=R.next();
  } ELSE{
    s1=s;
    r1=r;
    operand1 = {};
    operand2 = {};
    WHILE (joinable(r1, s)){
      operand1 = operand1 ∪ {s};
      s=S.next();
    }
    WHILE (joinable(r, s1)){
      operand2 = operand2 ∪ {r};
      r=R.next();
    }
    FOREACHs2 IN operand1 DO
      FOR EACH r2 IN operand2 DO
        OUTPUT join(s2, r2);
  }
}

```

We will analyze the average runtime complexity in an exercise at the book webpage <http://www.ifis.uni-luebeck.de/~groppe/SemWebDBBook/>.

6.5.2.1 Merge Join and Sideways Information Passing

When using SIP strategies, we can use the *next(lowerLimit)* method to retrieve directly an element equal to or larger than *lowerLimit*. The merge join can obviously benefit from the SIP strategy, because for unequal solutions of both operands, we already know that the larger solution *lowerLimit* is the lower limit for the next solution of the other operand retrieved by calling *next(lowerLimit)*. The following pseudo code contains the SIP version of the merge join algorithm, where SIP related code is marked with boldface:

```

S.open();
R.open();
s=S.next();
r=R.next();
WHILE(s!=null && r!=null){
    IF(s < r){
        s=S.next(r);
    } ELSE IF(r < s){
        r=R.next(s);
    } ELSE {
        s1=s;
        r1=r;
        operand1 = {};
        operand2 = {};
        WHILE(joinable(r1, s)){
            operand1 = operand1  $\cup$  {s};
            s=S.next();
        }
        WHILE(joinable(r, s1)){
            operand2 = operand2  $\cup$  {r};
            r=R.next();
        }
        FOR EACH s2 IN operand1 DO
            FOR EACH r2 IN operand2 DO
                OUTPUT join(s2, r2);
    }
}

```

When using the iterator versions of the operators, a SIP information may be passed through many operators until an index scan for retrieving the solutions of a triple pattern is reached. The index scan then can use the *next(lowerLimit)* method of a B^+ -tree to jump over possibly huge data. This will lead to enormous reduction in the runtime especially for large-scale datasets.

6.5.3 Index Join

The index join utilizes a given index of one of its join operands to optimize join processing: The index join iterates through the solutions of one join operand and

searches for relevant join partners, that is, solutions with the same bound values for the join variables, from the other join operand by using its index. An operand typically provides an index like a B⁺-tree if it is an index scan operator for retrieving the result of a triple pattern. We express the index join in pseudo code as follows:

```

FOR EACH  $s$  IN  $S$  DO
  FOR EACH  $r$  IN  $index(R, s)$  DO
    OUTPUT  $join(s, r)$ ;

```

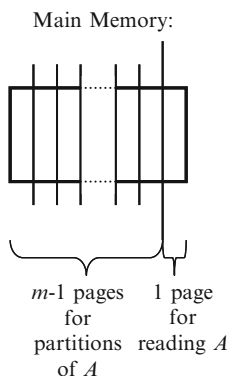
The function $index(R, s)$ is a function using a given index of the operator R to determine relevant join partners for the solution s . This join algorithm is typically used whenever one operand R provides an index and the solutions of the other operand S are not sorted according to the join variables, such that a merge join cannot be used without a preceding sorting phase. Let $g(|R|)$ be the costs of an index access on data with size $|R|$, then the index join has the runtime complexity $O(|S| * g(|R|))$ under the assumption that the data R has the property that $index(R, s)$ never returns large sets of data, but only one or few solutions. Theoretically, $g(|R|)$ is in $O(\log(|R|))$ for B⁺-trees, but practically B⁺-trees are typically quite flat with a height below 5 even for large data; that is, an index access is done in (nearly) constant time. Therefore, we can achieve a (nearly) linear complexity $O(|S|)$ for the index join for such kind of data, which is often the case for real-world data.

If both operands of a join are not index scans, but their solutions fit into main memory, then it is also efficient to first index the results of one operand R using an in-memory hash table and then apply the index join using the just created index. For indexing the solutions of R , a hash function over the join variables must be used, which maps bound values of the join variables in a solution to an integer, and such that the join partners for the solutions of S can be found within one index access. This type of index join is often also called *in-memory hash join*. The in-memory hash join is very efficient, because creating the index can be done in linear time to the results of R , that is, $O(|R|)$, and retrieving the join partners of R can be done in constant time. Under the assumption that we only have one or a few join partners for each solution of S in R , the average runtime complexity for joining is $O(|S|)$ and thus the overall average runtime complexity is $O(|R| + |S|)$.

6.5.4 Hash Join

We have presented the in-memory hash join, but we now describe the hash join for large data. Like the in-memory hash join, we use again hash functions over the join variables. The large data hash join first distributes the input data into smaller partitions using hash functions over the join variables. After the partitions become small enough, a general-purpose join algorithm, like the block-based nested-loop join, is used to join corresponding partitions of both operands.

Fig. 6.8 Pages in main memory for distributing the solutions of the operand A during the partitioning phase



In detail, we first determine the operand A with fewer solutions: If the sizes of the operands' solutions are not known, then the query optimizer can estimate which operand has fewer solutions. The results of the operand A is then partitioned into several *partitions*, that is, the solutions with the same result using the hash function are stored in the same partition. If the hash join operator can obtain m pages in main memory for its computations, then the hash join uses $m - 1$ partitions stored in $m - 1$ pages in main memory, and the remaining one page for reading in the data to be distributed (see Fig. 6.8). If one page of these $m - 1$ pages for the partitions is filled up, then the page is swapped to disk. When this partition round is finished, then the (not necessarily full) pages in main memory are stored on disk.

If one partition is larger than $m - 1$ pages, then another partition round for this large partition is performed using another hash function over the join variables. The process is repeated, until the size of each partition of A is less than m pages. For the reason of finishing the partitioning phase more early, we first distribute the solutions of the operand A , which has fewer solutions. We now distribute the solutions of the other operand B into partitions using the same hash functions as for A and in the same way as for A : If a partition was distributed for A , then we redistribute the partition for B using the same hash function, too. Unlike the partitions of A , the partitions of B do not need to fit into $m - 1$ pages. In the joining phase, we first load $m - 1$ pages of A into main memory and join them with the pages of B 's corresponding partition one by one by a block-based nested-loop join. Figure 6.9 presents an overview of the different phases of the hash join approach. Two partitioning rounds are performed in Fig. 6.9.

This hash join is known to be the fastest general-purpose join algorithm whenever the input data are neither sorted nor an already existing index can be used for joining.

We will analyze the number of pages accessed by the hash join algorithm in an exercise at the book webpage <http://www.ifis.uni-luebeck.de/~groppe/SemWebDBBook/>.

However, the hash join algorithm as described before can run into an infinity loop whenever a partition of A does not become smaller during redistributions,

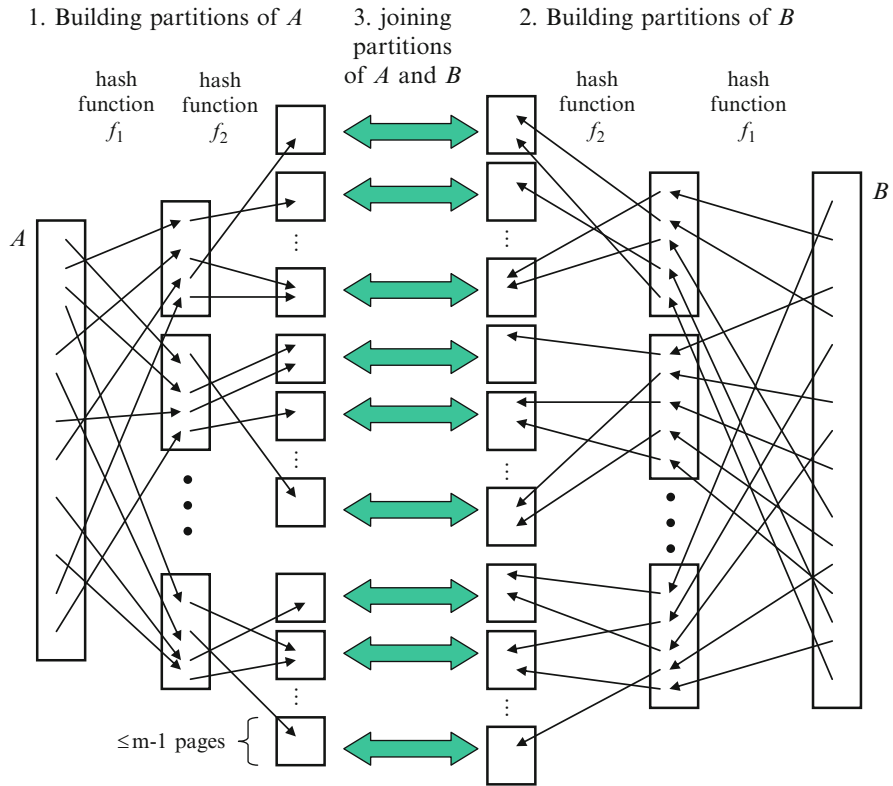


Fig. 6.9 Phases of hash join

as the solutions of this partition contain only the same bound values for the join variables. Fortunately, other general-purpose join algorithms than the block-based nested-loop join having the solutions of one operand completely in main memory can be used for joining this partition after several redistributions do not result in smaller partitions.

6.5.4.1 Hash Join and Sideways Information Passing

The hash join first reads in all solutions of one operand A for partitioning before any solutions of the other operand B is read. Thus, we already know the join partners, that is, solutions with the same bound values for the join variables, of the solutions of B in A and could filter out irrelevant solutions of B early, which do not have any join partners in A . Using all the join partners of A for this purpose is neither practical nor scalable because of a possibly large set of join partners. However, we can use a *bloom filter* for discarding irrelevant solutions early. The bloom filter uses a bit vector and sets those bits, which are returned by a given hash function for the

solutions of A . The bloom filter can be calculated during reading in the results of A . In our LUPOSDATE SPARQL engines, bloom filters are constructed in the SIP-FilterOperator (Iterator) operators. Afterward, the bloom filter can be attached to those operators in B , which bind values to join variables. These operators – typically index scans for determining the solutions of triple patterns – can now check by using the bloom filter if a corresponding join partner in B can exist. For determining the corresponding bit in the bit vector, the same hash function as when creating the bloom filter on the bound value of the join variables is applied to the solution of the triple pattern. If the bit is cleared, then we can surely discard this solution as it does not have any join partner in A . Due to the fact that hash functions may map several different values to the same integer, the bit for a solution without any join partners in A may be nevertheless set, such that we may have some *false drops*.

We have learned that a B^+ -tree offers the *next(lowerLimit)* method, which returns the next value equal to or larger than *lowerLimit* for a prefix search. B^+ -trees can optimize the application of the *next(lowerLimit)* method using interior nodes of the B^+ -tree. However, a bloom filter does not contain the information for such *lowerLimit* parameter, but we can determine a lower limit of the distance from the current value. We assume that the hash function h was used to construct a bloom filter. For example, we discard a solution with an id 10 bound to the join variable, because in our example, $h(10)$ returns a bit position, which is cleared in the bloom filter. Furthermore, the bits $h(11)$, $h(12)$, until $h(50)$ are also cleared in the bloom filter, but the bit $h(51)$ is set in the bloom filter. Therefore, we know that a lower limit for a relevant solution not to be discarded has a value with id 51 or higher, and we can call *next(51)* to retrieve it. Thus, in more general, if a join variable contains a value with id $id1$ and the bit $h(id1)$ is cleared in the bloom filter, then we can determine $id2$, such that the bit $h(id2)$ is set in the bloom filter and no $id3$ exists, such that $id1 < id3 < id2$ holds and the bit $h(id3)$ is set in the bloom filter. Then, $id2$ can be used to jump over the solutions, which do not have any join partner in A , and to retrieve directly a solution, which possibly has a join partner in A , by calling *next(id2)*.

6.6 Dynamically Restricting Triple Patterns

This join approach has been especially developed for in-memory join computation and is a variant of the index join. We assume that the seven in-memory hash indices S , SP , SPO , SO , P , PO , and O are given. A naive way to compute the join of two triple patterns is first evaluating individual triple patterns separately and then to perform a join on their results using standard join algorithms. Figure 6.10 visualizes this naive approach. However, we can leverage the properties of RDF and SPARQL in order to compute joins more efficiently.

In this section, we present a new and efficient approach to computing the join of triple patterns: we compute joins by dynamically generating *more restrictive triple patterns* (see Definition 11 in the previous chapter).

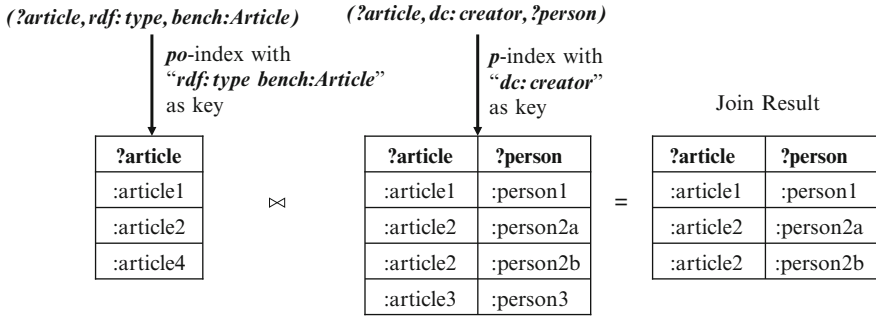


Fig. 6.10 A naive way to compute join

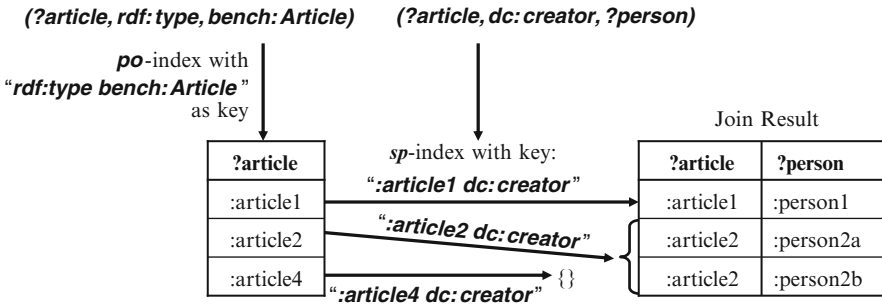


Fig. 6.11 Our approach to compute join

For example, $(?article \text{ rdf:type } bench:Article)$ is more restrictive than $(?article \text{ dc:creator } ?person)$. It is reasonable to assume that a more restrictive pattern typically retrieves less data than the less restrictive triple patterns.

In order to compute the join of two triple patterns, we compute the first triple pattern by means of the seven indices and then use the solutions of this triple pattern to replace the corresponding variables in the next triple pattern. In this way, we get a more restrictive triple pattern. By one index access, we can get the joined result of the current solution of the first triple pattern with the solutions of the second triple pattern directly and do not need to check the join condition. Figure 6.11 visualizes our approach to compute the join. If there are more triple patterns, we use the result of the already joined triple patterns to dynamically restrict the next triple pattern.

In this way, the join computation only involves the seven in-memory indices. Note that the seven indices only have to be generated once when reading the input data.

Example 2 (Dynamically generating more restrictive triple patterns). In order to evaluate the graph pattern


```

((( (?article rdf:type bench:Article)      AND
  (?article dc:creator ?person)          AND
  (?inproc rdf:type bench:Inproceedings) AND
  (?inproc dc:creator ?person2)         AND
  (?person foaf:name ?name)             AND
  (?person2 foaf:name ?name2))
  FILTER (?name=?name2))
  PROJ{?person, ?name} DISTINCT

```

over the following input RDF graph:

```

D={
  (:article1, rdf:type, bench:Article),      (t1)
  (:article2, rdf:type, bench:Article),      (t2)
  (:article4, rdf:type, bench:Article),      (t3)
  (:article1, dc:creator, :person1),         (t4)
  (:article2, dc:creator, :person2a),        (t5)
  (:article2, dc:creator, :person2b),        (t6)
  (:article3, dc:creator, :person3),         (t7)
  (:inproc1, rdf:type, bench:Inproceedings), (t8)
  (:inproc1, dc:creator, :person1),         (t9)
  (:person1, foaf:name, "Hans Fortune")     } (t10)

```

we first determine the solutions of the first triple pattern (*?article rdf:type bench:Article*) by using the *po* index with key “*rdf:type bench:Article*”. The triples (*t1*), (*t2*), and (*t3*) are returned for this index access and thus the result of the first triple pattern is $\langle \{(?article, :article1)\}, \{(?article, :article2)\}, \{(?article, :article4)\} \rangle$. We then bind each value of *?article* in the resultant set to the same variable in the second triple pattern (*?article dc:creator ?person*) and construct three more restricted triple patterns: (*article1 dc:creator ?person*), (*article2 dc:creator ?person*), and (*article4 dc:creator ?person*). For each new triple pattern, we use the *sp* index with the sequence of the subject and predicate literals as key. Therefore, the join of the first and second triple pattern is $\langle \{(?article, :article1), (?person, :person1)\}, \{(?article, :article2), (?person, :person2a)\}, \{(?article, :article2), (?person, :person2b)\} \rangle$.

The third triple pattern (*?inproc rdf:type bench:Inproceedings*) does not have any common variables with the join result of the first and the second triple patterns. We evaluate this triple pattern by using the *po* index with “*rdf:type bench:Inproceedings*” as key and get the solution $\langle \{(?inproc, :inproc1)\} \rangle$. Then, we compute the Cartesian product of the join of the first and second triple patterns and the result of the third triple pattern and get the join result of the first three triple patterns: $\langle \{(?article, :article1), (?person, :person1), (?inproc, :inproc1)\}, \{(?article, :article2), (?person, :person2a), (?inproc, :inproc1)\}, \{(?article, :article2), (?person, :person2b), (?inproc, :inproc1)\} \rangle$. We analogously proceed with the remaining triple patterns. The overall result of the whole SPARQL query is $\langle \{(?person, :person1), (?name, "Hans Fortune")\} \rangle$.

It is obvious that the given order for join computation is not optimal as the costly Cartesian product should be applied later or even avoided. Recall that the previous chapter introduced several approaches to optimizing the join order.

6.7 Sorting Numbering Scheme

We describe our specialized join approach for large-scale RDF datasets in this section. Since RDF data are modeled as a set of triples, six collation orders SPO, SOP, PSO, POS, OSP, and OPS are sufficient for sorting RDF data in any order. Therefore, recent approaches (e.g., [Neumann and Weikum 2008, 2009](#); [Weiss et al. 2008](#)) use six indices, each for one collation order, to manage RDF data. This storage schema allows for quick and scalable general-purpose query processing.

Furthermore, these approaches also adopt the technique of dictionary encoding ([Abadi et al. 2007](#)) to map RDF terms into integer identifiers (ids). Therefore, id triples are stored in six indices rather than the original literal triples. Actually, the sort criterion in evaluation indices of the other RDF stores such as *Hexastore* ([Weiss et al. 2008](#)) and *RDF3X* ([Neumann and Weikum 2008, 2009](#)) as well as our SPARQL engines are according to these ids rather than the RDF terms. Thus, operators like merge joins also use sort criteria according to the ids rather than according to the RDF terms themselves. Only a sort operator following the SPARQL specification ([Prud'hommeaux and Seaborne 2008](#)) requires a sort criterion according to the RDF terms instead of the ids. The main purpose of using integer ids to replace RDF terms is for compressing the RDF store in these approaches.

However, we also find another important application for these integer ids: they can be used to fast sort solutions. Having the capability of fast sorting, query processing, like computation of joins and elimination of duplications, can be performed more efficiently over large datasets. When we use these ids for sorting solutions, we call them *presorting numbers*.

6.7.1 Joins Without Presorting Numbers

For example, we have a SPARQL query with the three triple patterns TP1 (`?a <origin> <DLC>`), TP2 (`?a <records> ?c`), and TP3 (`?c <type> ?b`). When evaluating the query, we can first join the triple patterns TP1 and TP2 over the variable `?a`, or TP2 and TP3 over `?c`, or TP1 and TP3. The third alternative, the join between TP1 and TP3, is actually a Cartesian product having high costs, and thus we do not consider this join ordering further. State-of-the-art database management systems use selectivity estimations in order to determine the best join ordering, which we discuss in previous sections.

In order to perform a join on `?a` between TP1 and TP2, the existing approaches, *Hexastore* ([Weiss et al. 2008](#)) and *RDF3X* ([Neumann and Weikum 2008, 2009](#)), compute TP1 (`?a <origin> <DLC>`) according to the POS (or OPS) collation order and get the result sorted according to `?a`; compute TP2 (`?a <`

$records > ?c$) according to PSO and get the result sorted also according to $?a$. Consequently, a merge join can be directly used to compute the join of TP1 and TP2 over $?a$. The second join is computed between the results of the first join and of the remaining triple pattern TP3 ($?c < type > ?b$), and thus the join variable is $?c$. Since the result of the first join is sorted according to $?a$, the second join is computed using the hash join (see Fig. 6.12a).

Likewise, a merge join can be directly used for the join of the triple patterns TP2 ($?a < records > ?c$) and TP3 ($?c < type > ?b$), but cannot be directly used for the succeeding join between the results of the first join and that of the remaining triple pattern TP1 ($?a < origin > <DLC>$).

Sorting is time-consuming. It is a well-known fact that any sorting algorithm based on comparing and exchanging values needs at least $O(N \log N)$ steps, where N is the number of elements to be sorted. Therefore, instead of sorting data for performing a merge join, these approaches typically choose a hash join algorithm. A hash join does not require sorted input data and is usually faster than a normal merge sort join requiring an additional sorting phase. The hash join is simple and efficient when at least one of two operands of it fits into main memory. If neither of the two operands of a hash join fits into memory, the disk-based hash join algorithms become expensive (see Elmasri and Navathe 2000; Garcia-Molina et al. 2002). Our experiments show that a merge join with our fast sorting phase based on the presorting numbers outperforms significantly hash joins for large data.

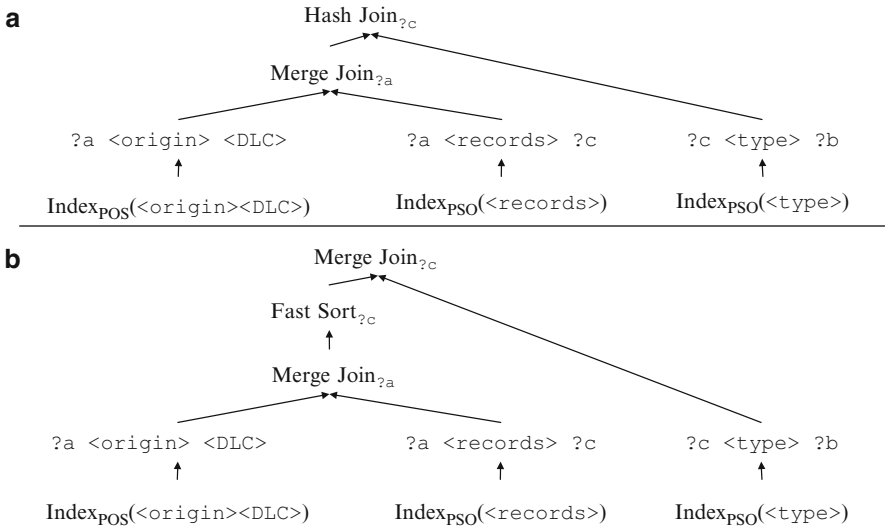


Fig. 6.12 Join computation of a nonbushy query with three triple patterns (a) without and (b) with using fast sorting

6.7.2 Joins with Presorting Numbers

We take as example the same query as in the previous section. Like RDF3X and Hexastore, we use a merge join to compute the first join between, for example, TP1 ($?a < origin > < DLC >$) and TP2 ($?a < records > ?c$) over $?a$. If we use the ids of RDF terms as presorting numbers, we can fast sort the result of this join according to $?c$. We use a variant of bucket sort (Knuth 1998) specialized to external sorting and also able to sort according to order criteria with several variables, as we will explain in later subsections. Consequently, the second join can be computed using a merge join instead of a hash join.

Figure 6.13 illustrates our fast sorting technique using the ids as presorting numbers. If the RDF data contain n different literals, then each literal can be mapped into an integer id in $[1, n]$. In order to sort solutions, we use n buckets, numbered from 1 to n . We describe in later subsections how to reduce the number of buckets.

During computing the first join between TP1 ($?a < origin > < DLC >$) and TP2 ($?a < records > ?c$), once a binding has been determined as a solution of this join, it is put into the corresponding bucket according to the id value of $?c$. Once the join computation is finished, the result in all these buckets has been sorted according to $?c$ and can be retrieved by accessing their contents in the order of the buckets.

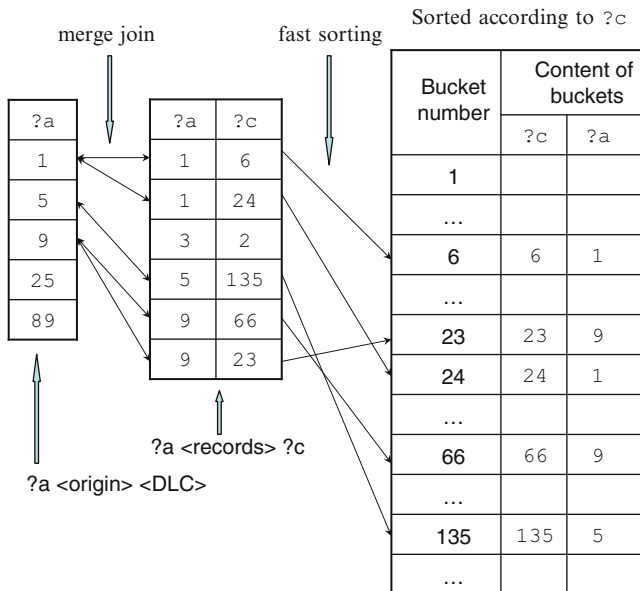


Fig. 6.13 Fast sorting according to $?c$ using the ids of RDF terms as presorting numbers when computing join

The last triple pattern $?c < type > ?b$ is computed using the index PSO, and thus its result is also sorted according to $?c$. Therefore, the second join between the results of the first join and of the last triple pattern can be computed using a merge join (see Fig. 6.12b).

6.7.3 Optimization of Fast Sorting

Usually, the presorting numbers (ids) of solutions are located at a certain range $[m, k]$, where $1 \leq m \leq k \leq n$, rather than dispersed over the whole id space. In this case, we need only $k + 1 - m$ buckets for sorting. We can determine the range $[m, k]$ during cardinality estimation of the results of triple patterns and of joins in the logical optimization phase. In the logical optimization phase, we can therefore store the minimum m and the maximum k of the range at the operator for the corresponding triple pattern in the execution plan.

If the range of the presorting numbers is relatively small, then we can store the buckets in main memory by simply using, for example, an array to store the solutions. If the range of the presorting numbers is large, then we have to store the buckets in an external storage like a hard disk. One way is to store each bucket in a single file. However, the number of solutions stored in a bucket is typically small and often even 1. Managing a large number of buckets with little content is inefficient.

An alternative without this disadvantage for sorting solutions is to divide the whole range of the presorting numbers into m smaller ranges. We then use m merge sorts for the m smaller ranges, each of which employs a heap for replacement selection, to sort the results with large range. We describe this approach in Fig. 6.14. In this way, we use much less buckets to store all sorted results. Our experiments show that using this way to sort data is very fast. In our experiments, we have used 1,000 ranges and heaps of size 256.

However, the ids of solutions are usually not continual integers, that is, there are gaps among these ids. This might lead to a bad distribution among m smaller ranges when using equal range sizes. This can be solved by using histograms of triple patterns.

For each kind of triple patterns, an equi-depth histogram is constructed. Among other information, each interval in this histogram contains the range and the number of triples allocated in this interval. Equi-depth histograms have the property to divide the data in such a way that each interval has the same or at least similar numbers of triples. Therefore, the intervals in the equi-depth histograms can be directly used as the smaller ranges into which the whole range of the presorting numbers is divided. In this way, we can get a perfect distribution among smaller ranges.

6.7.4 Sorting for Complex Joins

So far, the joins we consider have only one join partner, that is, only one common variable between two triple patterns. In most cases, the joins have only one join

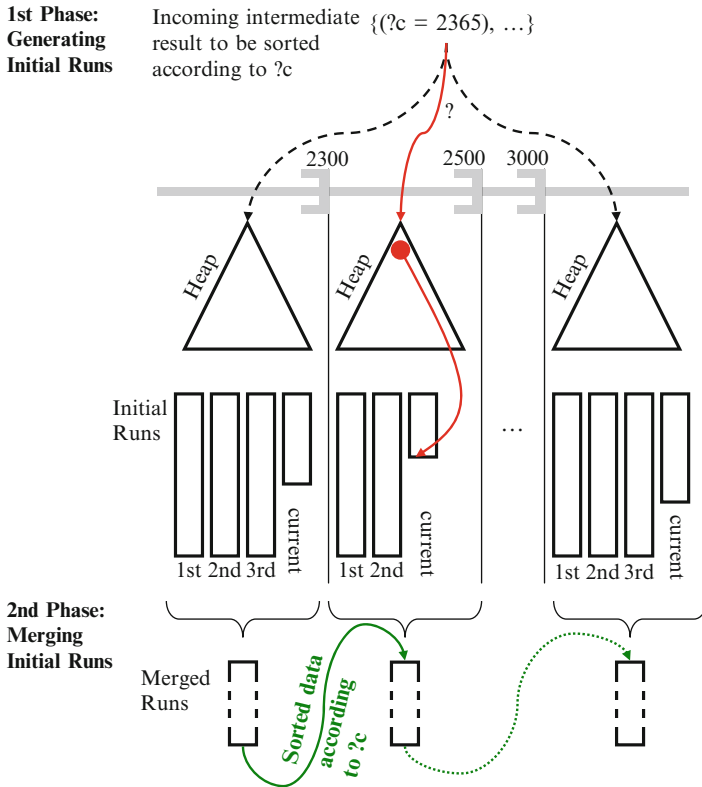
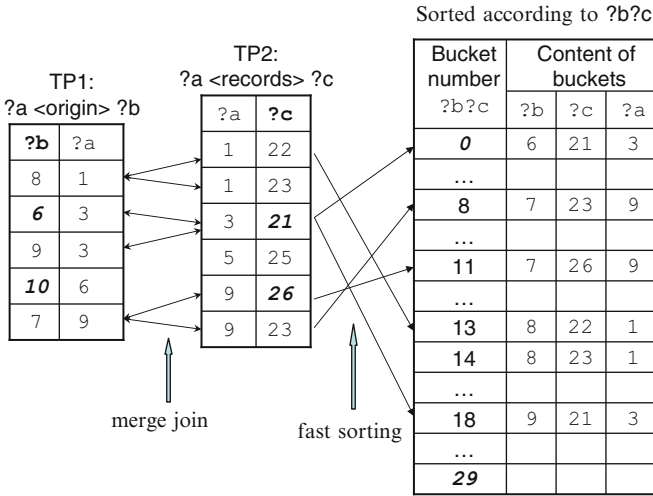


Fig. 6.14 Using multiple merge sorts with replacement selection to fast sort data with large range

partner, even if there are many joins in a query. Nevertheless, our fast sorting approach described so far can be extended to handle the joins with arbitrary numbers of join partners. The idea is to combine several presorting numbers to a unique presorting number.

For example, the execution plan for the three triple patterns TP1 ($?a < origin > ?b$), TP2 ($?a < records > ?c$), and TP3 ($?c ?b < text >$) is first to compute the join with the join partner $?a$ between TP1 and TP2 and then to compute the join with the two join partners $?b$ and $?c$ between the results of the first join and of TP3. In order to use the merge join to compute the second join, the result of the first join needs to be sorted according to both variables $?c$ and $?b$.

Figure 6.15 demonstrates how to sort the join result of TP1 and TP2 according to $?b$ as the primary order and $?c$ as the secondary order, denoted by $?b ?c$, while computing the join. In order to sort the join result according to $?b ?c$, we use (1) the id values of $?b$ returned by TP1, denoted as $TP1_{?b}$; and (2) the id values of $?c$ returned by TP2, denoted as $TP2_{?c}$. Let $MAX_{TP1, ?b}$ and $MIN_{TP1, ?b}$ be



$$\begin{aligned}
 ?b?c &= (TP1_{?b} - MIN_{TP1, ?b}) * (MAX_{TP2, ?c} - MIN_{TP2, ?c} + 1) + (TP2_{?c} - MIN_{TP2, ?c}) \\
 &= (TP1_{?b} - 6) * (26 - 21 + 1) + (TP2_{?c} - 21)
 \end{aligned}$$

Fig. 6.15 Sorting the result of join of TP1 and TP2 according to ?b (primary order) and ?c (secondary order)

the maximal and minimal values of $TP1_{?b}$, and $MAX_{TP2, ?c}$ and $MIN_{TP2, ?c}$ be the maximal and minimal values of $TP2_{?c}$. We use the formula,

$$\begin{aligned}
 &(TP1_{?b} - MIN_{TP1, ?b}) * (MAX_{TP2, ?c} - MIN_{TP2, ?c} + 1) + (TP2_{?c} - MIN_{TP2, ?c}) \\
 &= (TP1_{?b} - 6) * (26 - 21 + 1) + (TP2_{?c} - 21),
 \end{aligned}$$

to sort and compute unique presorting numbers of the join result between TP1 and TP2. Note that this formula is very similar to the function used in compilers for mapping an entry of a multidimensional array to a memory address.

Proposition 1. *The formula,*

$$(TP1_{?b} - MIN_{TP1, ?b}) * (MAX_{TP2, ?c} - MIN_{TP2, ?c} + 1) + (TP2_{?c} - MIN_{TP2, ?c}),$$

generates the unique combined presorting numbers and defines ?b as the primary sort criterion and ?c as the secondary sort criterion. The minimal combined presorting number is 0 and the maximal combined presorting number is

$$(MAX_{TP1, ?b} - MIN_{TP1, ?b}) * (MAX_{TP2, ?c} - MIN_{TP2, ?c} + 1) + (MAX_{TP2, ?c} - MIN_{TP2, ?c}).$$

Proof Sketch. The first part of the formula, $(TP1_{?b} - MIN_{TP1, ?b})$, is used to compute the number of values of ?b. Each value of ?b, that is, $TP1_{?b, ?b}$, can be combined with

any $TP2_{?c}$, and thus the number of possible combinations is the second part $*$ $(MAX_{TP2,?c} - MIN_{TP2,?c} + 1)$. The first and second parts together $(TP1_{?b} - MIN_{TP1,?b}) * (MAX_{TP2,?c} - MIN_{TP2,?c} + 1)$ allocate $TP1_{?b} - MIN_{TP1,?b}$ spaces. Each space has the size $MAX_{TP2,?c} - MIN_{TP2,?c} + 1$ and is for one $TP1_{POS}$ to combine with any $TP2_{?c}$. This part is also used to sort $?b$. The third part $+$ $(TP2_{?c} - MIN_{TP2,?c})$ sorts $?c$ in each space. In the space where one $TP1_{?b}$ is located, the possible maximal combined presorting number is $max_p = (TP1_{?b} - MIN_{TP1,?b}) * (MAX_{TP2,?c} - MIN_{TP2,?c} + 1) + (MAX_{TP2,?c} - MIN_{TP2,?c})$. In the space where $1 + TP1_{?b}$ is located, the possible minimal combined presorting number is $min_{p+1} = (1 + TP1_{?b} - MIN_{TP1,?b}) * (MAX_{TP2,?c} - MIN_{TP2,?c} + 1)$. Since $min_{p+1} - max_p = 1$, $TP1_{?b} - MIN_{TP1,?b}$ spaces are disjoint and the combined presorting numbers are unique.

Proposition 2. *In general, in order to sort data according to n variables, we need n types of presorting numbers P_1, \dots, P_n , where the maximal presorting numbers are $MAX_{P_1}, \dots, MAX_{P_n}$, and all the minimal presorting numbers are 0. If the range of the original presorting numbers p is in $[MIN, MAX]$, then we compute a new $p' = p - MIN$, and thus the minimal value of p' is 0. We can combine these presorting numbers and retrieve the unique combined presorting numbers in $n!$ different ways. For example, the formula, $(\dots((P_1 * (MAX_{P_2} + 1) + P_2) * (MAX_{P_3} + 1) + P_3) * (\dots(MAX_{P_n} + 1)) + P_n$, computes the unique presorting numbers and defines P_1 as the primary sort criterion, P_2 as the secondary sort criterion, \dots, P_n as the n th sort criterion. The maximum of the combined presorting numbers is $(\dots((MAX_{P_1} * (MAX_{P_2} + 1) + MAX_{P_2}) * (MAX_{P_3} + 1) + MAX_{P_3}) * \dots) * (MAX_{P_n} + 1) + MAX_{P_n}$.*

Proof Sketch. According to Proposition 1, the most inner part, $P_1 * (MAX_{P_2} + 1) + P_2$, generates the unique presorting numbers of the combined P_1 and P_2 , and defines P_1 as the primary sort criterion and P_2 as the secondary sort criterion, denoted by P_1P_2 ; $(P_1P_2 * (MAX_{P_3} + 1) + P_3)$ generates the unique presorting numbers of the combined P_1, P_2 , and P_3 , and defines P_1P_2 as the primary sort criterion and P_3 as the secondary sort criterion, denoted by $P_1P_2P_3$. We proceed analogously until we get the unique presorting number of the combined P_1, \dots, P_n . In this way, we can prove the correctness of the formula.

The $MAX - MIN$ range can grow quickly when processing a join with multiple join partners. However, a large range of presorting numbers is not a problem for our fast sorting approach, because we divide the whole range of the presorting numbers into m smaller ranges and (merge) sort the ranges independently from each other. Therefore and based on the experience in our experiments, we neither need more space nor more time for a larger range, if the number of data to be sorted is the same.

6.7.5 Additional Benefits from SIP Strategies

A merge join looks for the equivalent values from two sorted operands by comparing their results pairwise. If two values are unequal, the merge join continues

reading the following data from the operand with the smaller value until an equal or larger value is seen. When processing very large datasets, reading each data in turn for finding a certain value is quite time-consuming.

An improvement is using the SIP strategy as described in (Neumann and Weikum 2009). SIP applies the information of the larger value L to the side of the operand with the smaller value for directly going to the data, which is equal to or larger than L . In particular, when data are stored using B^+ -trees, the larger value L can be used as key for directly finding the wanted leaf. Using SIP allows jumping over big gaps by accessing interior nodes of the B^+ -trees and thus avoiding searching along a possibly long chain of leaves.

For hash joins, the SIP strategy can use bloom filters. A bloom filter is a bit vector, which is created by applying a hash function to one operand. This bit vector is afterward used to filter out irrelevant data of the other operand. The bloom filter can also be used to compute the number of distinct values, which can be jumped over. The upper bound for this number corresponds to the number of unset bits in the bloom filter. However, since a hash function might map many distinct values to a same integer, the real number of distinct values to be jumped over can be (much) larger. Furthermore, the number of distinct values is at most the length of the bit vector of the bloom filter. Therefore, using bloom filters is not scalable.

Having our fast sorting capability, the efficient merge join can be applied instead of the hash join. Furthermore, using SIP merge joins can jump over bigger gaps than hash joins. Therefore, the application of our sorting numbering scheme in combination with SIP can significantly speed up query processing.

6.8 Optional

The Optional operator returns the result of a join between its operands and additionally all not joined solutions of its left operand. Therefore, we can modify any join algorithm to return joined solutions and additionally not joined solutions of the left operand. We do not provide adapted algorithms computing the result of an Optional operation for all discussed join algorithms here and leave this to the reader, but present the adapted algorithm of the merge join algorithm using SIP in the next subsection.

6.8.1 MergeOptional

Assuming S to be the left and R to be the right operand, we modify the merge join algorithm using SIP to calculate the result of an OPTIONAL construct in the following way:

```

S.open();
R.open();
s=S.next();
r=R.next();
WHILE(s!=null && r!=null){
  IF(s < r){
    OUTPUT s;
    s=S.next();
  } ELSE IF(r < s){
    r=R.next(s);
  } ELSE {
    s1=s;
    r1=r;
    operand1 = {};
    operand2 = {};
    WHILE(joinable(r1, s)){
      operand1 = operand1  $\cup$  {s};
      s=S.next();
    }
    WHILE(joinable(r, s1)){
      operand2 = operand2  $\cup$  {r};
      r=R.next();
    }
    FOR EACH s2 IN operand1 DO
      FOR EACH r2 IN operand2 DO
        OUTPUT join(s2, r2);
  }
}
WHILE(s!=null){
  OUTPUT s;
  s=S.next();
}

```

In this version of the algorithm, we return s as result whenever no join partners in R can be found. The modified code is marked with boldface.

6.9 Duplicate Elimination

SPARQL uses the modifier *DISTINCT* to require a result without duplicates. In this subsection, we describe different algorithms for the elimination of duplicates.

6.9.1 Duplicate Elimination Using Hashing

This version of duplicate elimination first partitions its input data by using hash functions until all partitions fit into main memory. Afterward, the algorithm can use any in-memory algorithm for duplicate elimination by, for example, determining the set of solutions in each partition. Sets can be determined from a sequence of

solutions containing duplicates by, for example, using a hash table or balanced trees like AVL trees.

6.9.2 Duplicate Elimination Using Sorting

Duplicate elimination using sorting just first sorts its solutions and afterward scans the sorted solutions, and returns only those solutions, which are different to their previous ones. In some query plans, the solutions are already sorted before the *DISTINCT* operator, such that the *DISTINCT* operator does not need to sort the data initially.

6.9.3 Duplicate Elimination Using Presorting Numbers

Fast sorting data using presorting numbers is of great benefit not only to merge joins, but also to many other operations in query processing. Using the presorting numbers, *DISTINCT* operations can be efficiently performed: when processing the operation just before the *DISTINCT* operation, once a solution is computed, it is sorted using the corresponding presorting numbers in the same way for sorting the join result as illustrated in Fig. 6.13. However, we only store one of the solutions with the same presorting number. When the operation before the *DISTINCT* operation is finished, its result will not contain the duplicates. Consequently, a separate *DISTINCT* operation is not needed anymore.

6.10 Cost Model

In the previous chapter, we have already described logical plan generation based on estimations of the result cardinality. The main goal there was to reduce the number of solutions as early as possible in the operatorgraph.

Different physical operators have different costs concerning CPU processing time, I/O costs for reading and storing solutions, as well as costs for used space in main memory or on disk. The physical plan generator takes all these costs into consideration, computes the total costs of possible physical operators, and chooses the physical operator with the best estimated total costs. As I/O costs are typically the lion's share in the total costs, that is, I/O operations are the slowest ones in a nondistributed computer system, most physical query optimizers focus on the I/O costs. Some query optimizers determine the I/O costs by estimating the number of solutions to be read and written by a specific operator. Other query optimizers compute the I/O costs by estimating the number of page accesses for a specific operator, which considers the block nature of I/O devices such as hard disks. Both variants produce similar good results, but cannot be mixed.

6.11 Performance Evaluation

For the performance evaluation, we consider two main scenarios.

In the first scenario, we use small to middle-sized datasets, which completely fit into main memory. In this scenario, our approaches for main-memory indexing and join processing promise optimal performance.

In the second scenario, we use large-scale datasets with over one billion triples. These datasets do not fit into main memory anymore and we comprehensively analyze the disk-based indexing (and joining) approaches.

6.11.1 Performance Evaluation for In-memory Databases

The SP²B benchmark (Schmidt et al. 2009) includes a set of 18 queries, which contain more features of SPARQL and address more optimization techniques than many other Semantic Web benchmarks such as the LUBM benchmark (Guo et al. 2005). The SP²B benchmark uses a data generator, which can generate data of different size. The SP²B data set imitates an RDF version of the real-world DBLP data set (Ley 2010); that is, the data structure of the SP²B data set is very similar to real-world data. Furthermore, the SP²B benchmark does not consider inference based on an ontology, which we do not consider here due to our focus on basic join algorithms. Therefore, we choose the SP²B benchmark in our experiments. In our figures, we present the average of ten execution times of reading the input and generating the indices, and of applying the SPARQL queries of the SP²B benchmark.

The test system uses an Intel Core 2 Duo CPU T7500 with 2.2 GHz, 2 GB main memory, Windows XP Professional 2002, and Java 1.6. We use Jena ARQ (Wilkinson et al. 2003) and Sesame (Broekstra et al. 2002) as SPARQL database engines since they support the current SPARQL version (Prud'hommeaux and Seaborne 2008), which is not fully supported by many other SPARQL processing engines. Furthermore, we have implemented an in-memory version of the approach presented in (Weiss et al. 2008), which we call *In-memory Hexastore* in the following paragraphs. In-memory Hexastore uses merge joins to join two triple patterns at the first level and standard relational join algorithms like index joins for succeeding joins. Our implementation of In-memory Hexastore reorders the join operands according to the result sizes of triple patterns.

In the figures, we call our approach *RestrictingTP*. We have measured different variants of our approach. *RestrictingTP-OrderSize* represents the execution times of our approach when reordering the triple patterns according to the result sizes of each triple pattern. *RestrictingTP-OrderVar* represents the execution times of our approach when reordering the triple patterns according to the restrictiveness of triple patterns. *RestrictingTP-OrderVarSize* represents the hybrid approach, which orders the triple patterns primarily according to the restrictiveness of triple patterns

and secondarily according to the result sizes of the triple patterns. We use hash maps for our seven indices, but use B⁺-trees whenever we mark the experiment with B⁺, where we also order the triple patterns primarily according to the restrictiveness and secondarily according to the result sizes of the triple patterns.

6.11.1.1 Index Construction Time

Reading the input data and constructing the indices only need to be done *once*. Afterward, the indices can be used to evaluate many queries. Thus, less time for query evaluation is more critical than less time for index construction. The index construction time for Sesame and Jena is the smallest (see Fig. 6.16). The index construction time for *RestrictingTP*, *RestrictingTP-orderSize*, *RestrictingTP-orderVar*, and *RestrictingTP-orderVarSize* is obviously the same, and thus we use the time for *RestrictingTP* for all of them, which are approximately two times slower than Jena and 3.6 times slower than Sesame. In-memory Hexastore and B⁺ perform the worst.

6.11.1.2 Query Evaluation

For all queries used in this experiment, the time to compile queries and perform the logical and physical optimization step is below one millisecond.

For each query of the SP²B benchmark (Schmidt et al. 2009), In-memory Hexastore performs worst, as In-memory Hexastore processes time-consuming searches in sorted lists for index accesses, which is avoided in our approach.

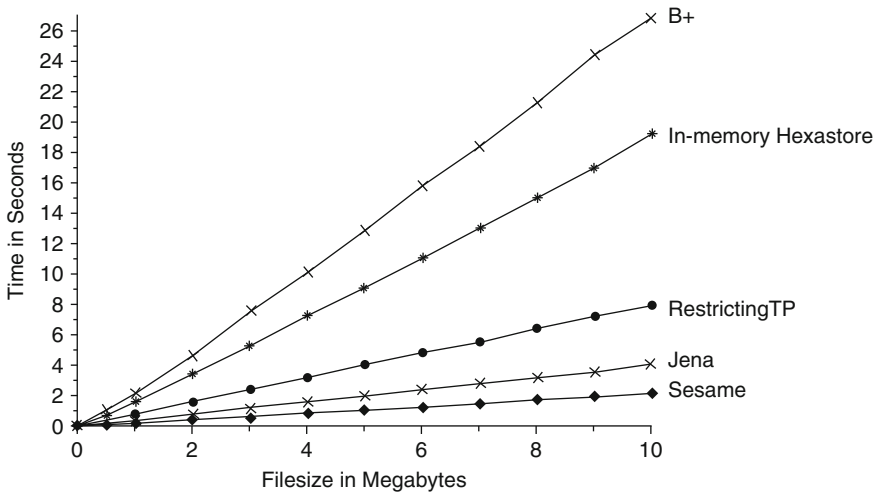


Fig. 6.16 Index construction time

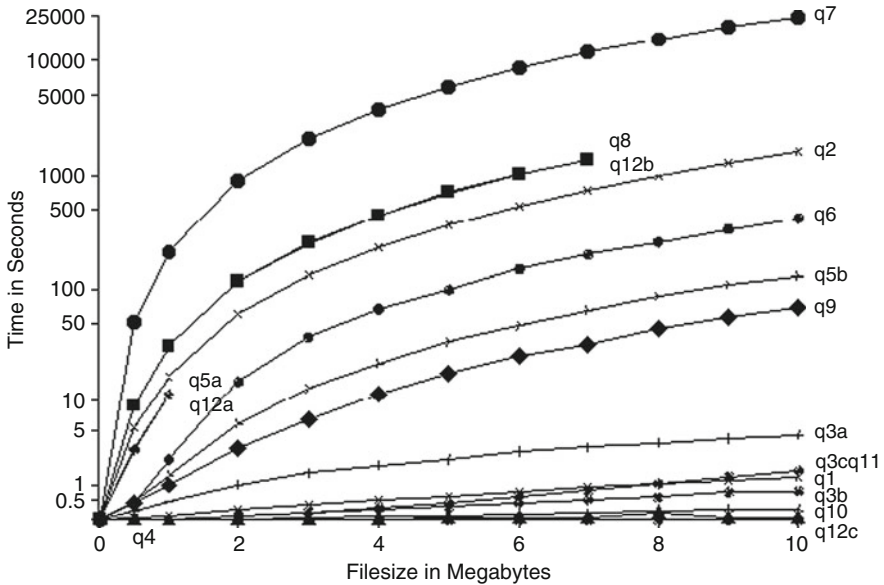


Fig. 6.17 Execution times of the queries q1–q12c of In-memory Hexastore

Furthermore, out-of-memory errors occur for the queries q4, q5a, and q12a. For simplicity of presentation, we present the execution times of In-memory Hexastore in an extra figure (see Fig. 6.17) and do not present them in the figures especially for the different queries.

Query q1. This query consists of three triple patterns, the overall result of which contains only one solution. All approaches except *In-memory Hexastore* need only less than 4 ms.

Query q2. This query consists of ten triple patterns, one of which is in an OPTIONAL construct. The result is additionally sorted. Sesame and Jena perform best (see Fig. 6.18). The implementation of sorting can be still much optimized in the benchmarked version of our SPARQL engine, such that we believe that our approach performs much better after these optimizations have been done.

Query q3a–c. These queries consist of two triple patterns and a comparison of a variable with a constant value. Due to space limitations, we present the execution times of only q3a here (see Fig. 6.19). For q3a, our approach performs best and then Sesame and Jena. For q3b and q3c, all approaches and query engines except of In-memory Hexastore need less than 100 ms, where *RestrictingTP-orderVarSize* performs best, the execution times of which are below 4 ms.

Query q4. This query consists of eight triple patterns with a less than value comparison (“<”) between two variables. Figure 6.20 shows that *RestrictingTP-orderVarSize* performs best. Jena is the slowest. *In-memory Hexastore*, *RestrictingTP*, and *RestrictingTP-orderSize* compute the triple patterns in an inefficient way, such that out-of-memory errors occur.

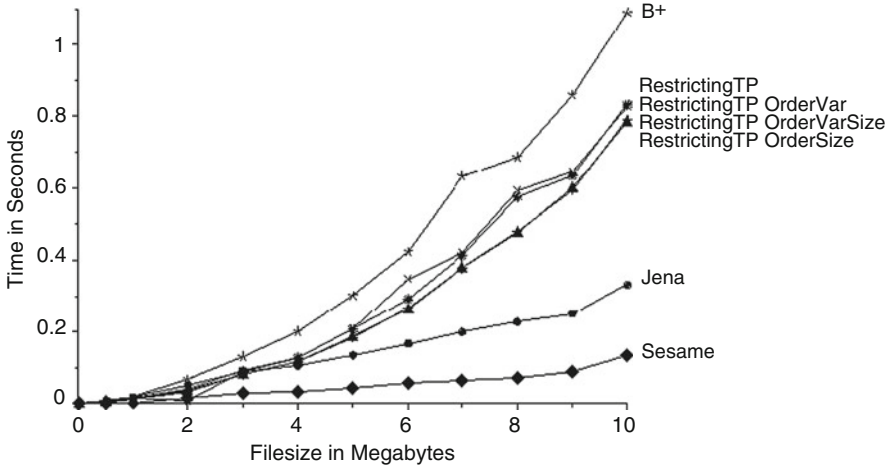


Fig. 6.18 Execution times of q2

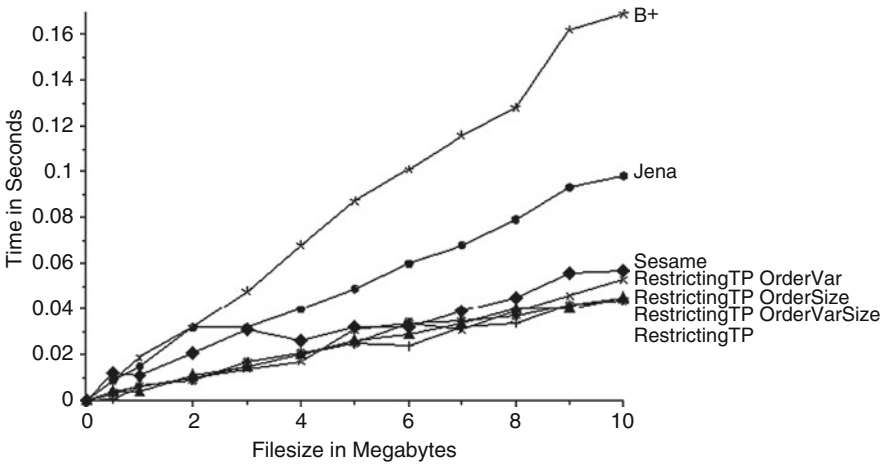


Fig. 6.19 Execution times of q3a

Queries q5a and q5b. q5b is a manually optimized query of q5a. Restricting-orderVar and Restricting-orderVarSize perform best for q5a (see Fig. 6.21). For q5b, Restricting-orderVarSize is slightly slower than Sesame (0.3 s), but 156 times faster than Jena.

Query q6. Our approaches are slightly slower than Sesame and Jena (see Fig. 6.22) since the complex OPTIONAL constructs with many filter expressions in q6. In the implementation of our prototype, there is still much space for optimizing this kind of queries.

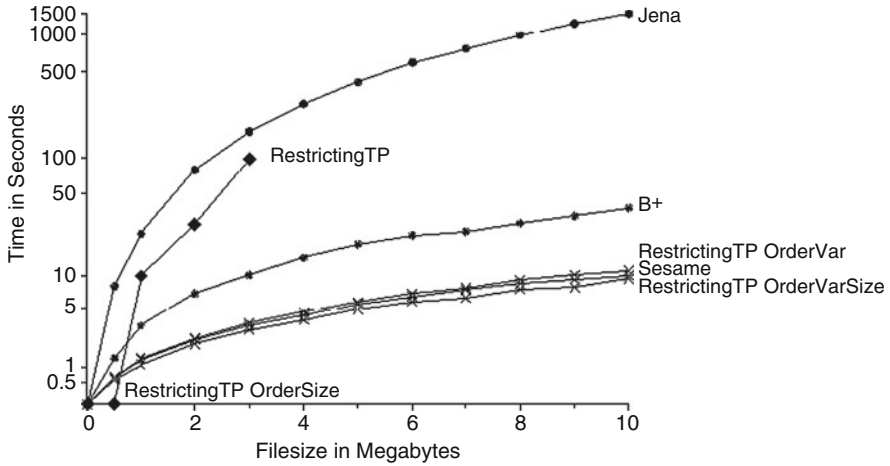


Fig. 6.20 Execution times of q4

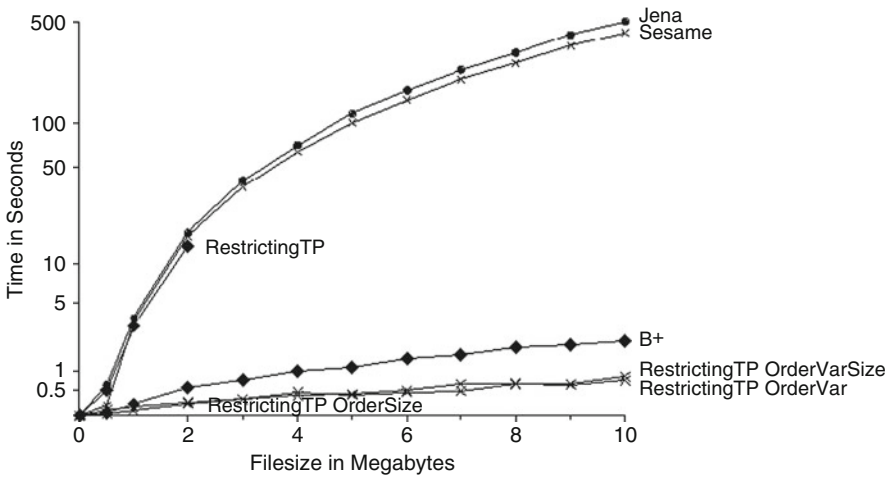


Fig. 6.21 Execution times of q5a

Query q7. q7 consists of several nested OPTIONAL constructs. Here, our approaches perform much better than the others (see Fig. 6.23).

Query q8 and q9. q8 and q9 contain common subexpressions in the operands of UNION, which is currently not optimized by our prototype. Therefore, the execution times of our prototype are higher than those of Sesame and Jena (see Fig. 6.24).

Queries q10 and q11: q10 and q11 consist of one triple pattern, the result of which is retrieved below 300 ms for all different approaches and query engines except of In-memory Hexastore.

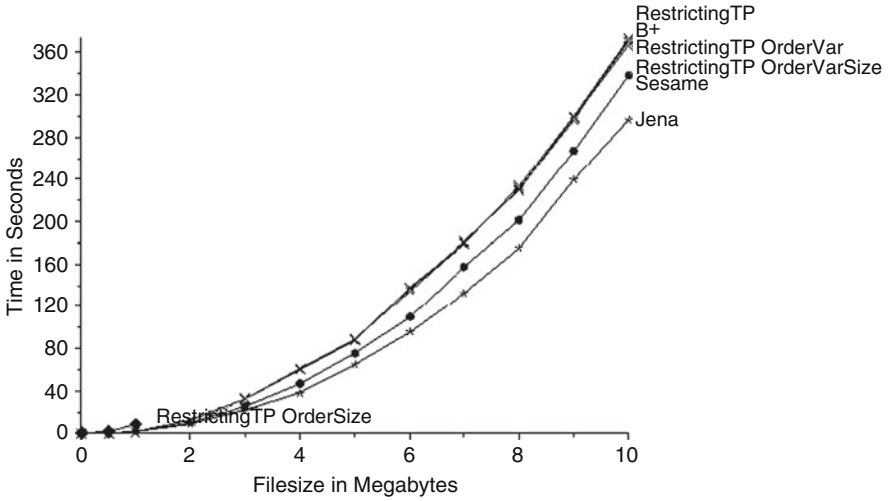


Fig. 6.22 Execution times of q6

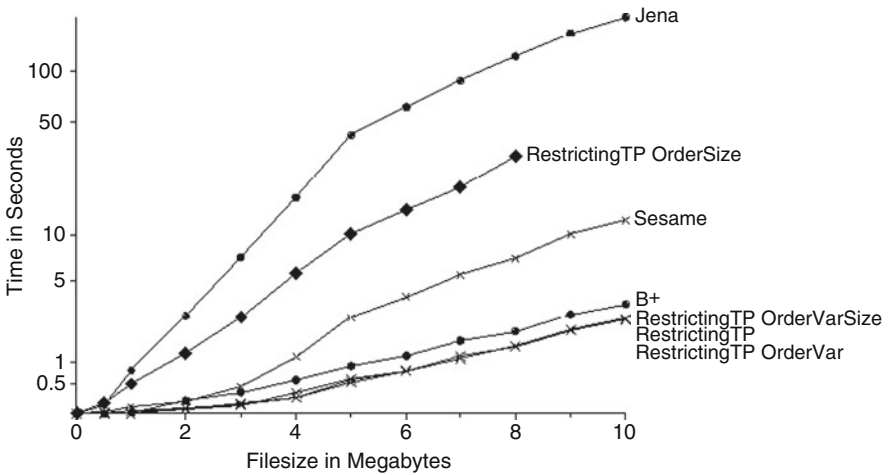


Fig. 6.23 Execution times of q7

Queries q12a–c. These queries contain the ASK construct; that is, they return true if there is some result and otherwise false. One optimization is to abort the evaluation of queries if there is at least one result. Our approaches perform similarly to the other approaches for q12c and are slightly slower for q12a and q12b.

Average execution times of all queries q1–q12c. Figure 6.25 shows the average execution times of all queries q1–q12c. Overall, our approaches and especially *RestrictingTP-OrderVarSize* are the fastest. *Sesame* is more than two times slower, and *Jena* is more than six times slower than *RestrictingTP-OrderVarSize*. In-memory Hexastore is the slowest approach.

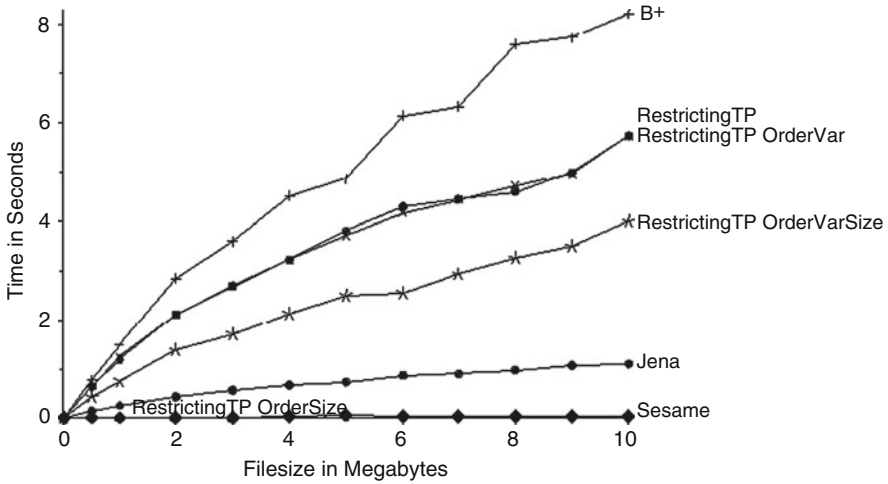


Fig. 6.24 Execution times of q8

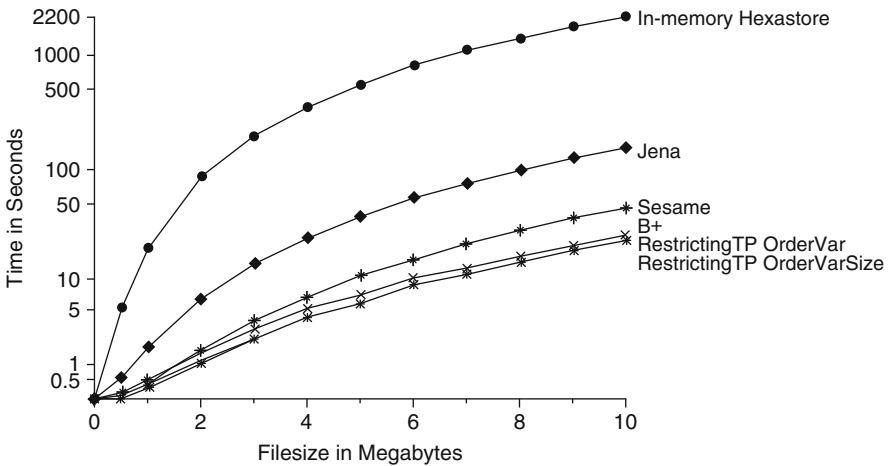


Fig. 6.25 Average execution times of all queries q1-q12c

6.11.2 Performance Evaluation for Large-Scale Datasets

We study the performance benefits for our sorting numbering scheme integrated into other index approaches. For these experiments, we focus on the index approach *RDF3X* in (Neumann and Weikum 2008, 2009), since it is similar to Hexastore in (Weiss et al. 2008), but uses a simpler and faster index structure than (Weiss et al. 2008). We compare the pure *RDF3X* approach, that is, using hash joins when data become unsorted, with the *RDF3X-Sort* approach, that is, our sorting numbering

scheme integrated into the *RDF3X* approach. *RDF3X-Sort* uses fast sorting for applying merge joins to replace hash joins and for elimination of duplicates.

The original *RDF3X* prototype (Neumann and Weikum 2008, 2009) has several limitations:

- It does not support full SPARQL 1.0.
- It only supports very simple filter expressions.
- It neglects prefixes in predicates. This improves the performance, but leads to information loss. For example, the two different triples ($\langle a \rangle$, $\langle \text{www.film}/\text{title} \rangle$, “Ratatouille”) and ($\langle a \rangle$, $\langle \text{www.game}/\text{title} \rangle$, “Ratatouille”) are stored as one triple ($\langle a \rangle$, $\langle \text{title} \rangle$, “Ratatouille”) in the original *RDF3X* prototype.
- It does not support data types, such that, for example, the two identical integer values +2 and 2 are treated as two different strings.
- It supports only in-memory hash joins; that is, if the operands of hash joins cannot fit into memory, *RDF3X* cannot process the hash joins.

In order to lift these limitations and avoid problems resulting from not supported features of the original *RDF3X* prototype, we have reimplemented the *RDF3X* approach. Our reimplementation successfully runs all the W3C test cases (Feigenbaum 2008), which contain over 200 queries. As we will show in the following subsections, the execution times of our reimplementation are similar to, and often outperform those of the original *RDF3X* prototype, compared with results of Neumann and Weikum (2009), although we have used Java as programming language and the *RDF3X* system is implemented in C++. An online demonstration of our implementations is publicly available (see Groppe and Groppe 2009; Groppe et al. 2009b).

The test system for the evaluation of queries uses a Dual Quad Core Intel CPU X5550 computer with 2.67 GHz, 6 GB main memory, Windows XP Professional (x64 Edition), and Java 1.6 64 bit. We have run the experiments ten times and present the average execution times as well as the standard deviation of the sample.

In order to build indices faster over the two very large datasets, index constructions are performed in a cluster with additional 6 Intel Core 2 Quad CPU Q9400 computers, each with 2.66 GHz, 4 GB main memory, Windows XP Professional (32 bit), and Java 1.6.

We use two large-scale datasets: UniProt (Swiss Institute of Bioinformatics 2009) and Billion Triples Challenge (BTC) (Semantic web challenge 2009), and corresponding queries. Three kinds of indices are constructed over the two datasets: two dictionary indices for mapping between RDF terms and integer ids; six evaluation indices according to the six collation orders of RDF for evaluating SPARQL queries; six histogram indices for fast generating histograms for triple patterns.

6.11.2.1 UniProt

UniProt (Swiss Institute of Bioinformatics 2009) is a comprehensive repository of protein sequence and annotation data. We have used the version 15.14 of the 9th February 2010 of it, which contains over 1.5 billion triples.

We have used the queries of Neumann and Weikum (2009), which we name UP 1 to UP 8. However, the number of results of these queries and especially the number of solutions for the hash joins are quite small. Therefore, we added some additional queries (EUP 1 to EUP 8) with bigger cardinalities, which are presented below.

Table 6.2 presents the types of operations of the UniProt queries performed by the pure *RDF3X* approach without using our fast sorting technique. In comparison, the *RDF3X-Sort* approach, that is, our sorting numbering scheme integrated into the *RDF3X* approach, uses fast sorting and merge joins instead of hash joins and optimizes duplicate elimination by also using our fast sorting technique. Table 6.1 presents the execution costs by the two approaches when they process these UniProt queries over 1.5 billion triples.

The queries UP 1 to UP 8 from *RDF3X* (Neumann and Weikum 2009) are favorable to *RDF3X*, since the number of solutions is quite small, and thus the simple and fast in-memory hash join can be used. Therefore, the performance gain is quite small by *RDF3X-Sort* for these queries. However, for queries with larger intermediate results, *RDF3X-Sort* is often about 10 times up to 32 times faster than the *RDF3X* approach (see EUPs 1, 2, 3, and 5).

The time for index construction for the over 1.5 billion triples was 57 h. The space consumption is 19.5 GB for the two dictionary indices, 47.1 GB for the six evaluation indices, and 47.1 GB for the six histogram indices.

Table 6.2 presents the types of operations of the UniProt queries performed by the pure *RDF3X* approach without using our fast sorting technique, when processing the UniProt queries. In the following paragraphs, we provide the additional queries for the UniProt dataset.

We assume that all UniProt queries define the namespaces *rdf*, *rdfs*, and *up* by

Table 6.1 Evaluation times (in seconds) for UniProt Data

Query	RDF3X	RDF3X-sort	RDF3X/ RDF3X-sort	Histogram computation
UP 1	0.0391 ± 0.0106	0.0375 ± 0.0174	1.043	0.14
UP 2	0.35 ± 0.021	0.35 ± 0.02	1	1.014
UP 3	0.925 ± 0.0702	0.8969 ± 0.0474	1.031	0.395
UP 4	0.321 ± 0.025	0.309 ± 0.013	1.039	0.233
UP 5	0.79 ± 0.04	0.79 ± 0.05	1	0.005
UP 6	0.55 ± 0.006	0.55 ± 0.01	1	0.02
UP 7	0.5453 ± 0.0047	0.5438 ± 0.0061	1.003	0.042
UP 8	0.8469 ± 0.0117	0.8515 ± 0.0203	0.995	0.286
EUP 1	27.320 ± 4.427	2.7468 ± 0.1044	9.946	2.061
EUP 2	3.1172 ± 0.1078	0.3391 ± 0.1081	9.193	0.572
EUP 3	1620.1 ± 5.98	49.6735 ± 0.236	32.615	1.614
EUP 4	1.55 ± 0.018	0.8657 ± 0.0102	1.79	1.05
EUP 5	0.2096 ± 0.0102	0.0172 ± 0.0046	12.186	0.595
EUP 6	2.5141 ± 0.0203	1.9937 ± 0.0077	1.261	0.7624
EUP 7	42.6124 ± 0.783	4.9063 ± 0.1245	8.685	0.5
EUP 8	0.2843 ± 0.0645	0.0376 ± 0.0187	7.561	0.55

```

PREFIX rdf:
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX up:   <http://purl.uniprot.org/core/>.

```

Query EUP 1:

```

select * where{
?x rdf:type up:Sequence_Conflict_Annotation;
  up:conflictingSequence ?y.
?y rdf:type up:External_Sequence.}

```

Query EUP 2:

```

select * where {
?c rdf:type up:Concept; rdfs:label ?l1;
  up:obsolete "true"; rdfs:subClassOf ?c2.
?c2 rdfs:label ?l2.}

```

Query EUP 3:

```

select * where {
?x up:sequenceFor ?y; rdf:type up:Sequence.
?y rdf:type up :Protein;up:reviewed "false";
  up:created "2009-07-28".}

```

Query EUP 4:

```

select * where {
?x up:date "1996"; rdf:type>?t.
?t rdfs:subClassOf ?c.}

```

Query EUP 5:

```

select * where {
?x up:cofactor "Iron"; rdfs:subClassOf ?c.
?c up:name ?n.}

```

Query EUP 6:

```

select * where {
?x rdf:type up:Tissue; rdfs:label ?l1;
  rdfs:seeAlso ?y.
?y rdfs:label ?l2; up:database "eVOC".}

```

Query EUP 7:

```

select distinct * where {
?x rdf:type up:Sequence_Conflict_Annotation;
  up:conflictingSequence ?y.
?y rdf:type up:External_Sequence.}

```

Query EUP 8:

```

select distinct * where {
?x up:cofactor "Iron"; rdfs:subClassOf ?c.
?c up:name ?n.}

```

Table 6.2 Operations by RDF3X for the UniProt queries

Query	Number of merge joins	Number of hash joins	DISTINCT
UP 1	2	1	
UP 2	10	2	
UP 3	10	1	
UP 4	8	2	
UP 5	5	2	
UP 6	11	1	
UP 7	11	1	
UP 8	11	1	
EUP 1	1	1	
EUP 2	3	1	
EUP 3	3	1	
EUP 4	1	1	
EUP 5	1	1	
EUP 6	3	1	
EUP 7	1	1	√
EUP 8	1	1	√

6.11.2.2 Billion Triples Challenge

The major part of the dataset of the Billion Triples Challenge (BTC) (Semantic web challenge 2009) was crawled during February/March 2009 based on datasets provided by, for example, Falcon-S, Sindice, Swoogle, SWSE, and Watson. We have imported *all* over 830 million distinct triples of the Billion Triples Challenge. In comparison, the performance analysis in Neumann and Weikum (2009) used only a subset of it.

Like the UniProt queries used by RDF3X (Neumann and Weikum 2009), the queries for BTC used by (Neumann and Weikum 2009) return very small intermediate and final results. Therefore, we also use some additional queries (EBTC 1 to EBTC 8) with bigger cardinalities (see below), as well as the ones of (Neumann and Weikum 2009) (BTC 1 to BTC 8). Table 6.4 presents the query operations performed by the original RDF3X approach, and Table 6.3 presents the processing times of these queries by the two approaches.

Although the queries BTC 1 to BTC 8 are designed to retrieve very small results and thus are favorable to RDF3X, the RDF3X-Sort still has a similar (or a slightly better) evaluation performance (up to 24%). When processing those queries with large intermediate and final results (EBTC 1 to EBTC 8), the RDF3X-Sort approach shows significant performance improvements and is up to several orders of magnitude better than the pure RDF3X.

The time for index construction for the BTC dataset with over 830 million distinct triples was 30 h. The space consumption is 31.1 GB for the dictionary indices, 30.8 GB for the evaluation indices, and 30.8 GB for the histogram indices.

Table 6.3 Evaluation times (in seconds) for BTC Data

Query	RDF3X	RDF3X-sort	RDF3X/ RDF3X-sort	Histogram computation
BTC 1	0.0469 ± 0.0155	0.045 ± 0.0049	1.042	23.5
BTC 2	0.0359 ± 0.0076	0.0359 ± 0.0105	1	7.419
BTC 3	0.3032 ± 0.0547	0.3032 ± 0.0443	1	2.642
BTC 4	39.3234 ± 0.123	38.0327 ± 0.0491	1.034	73.23
BTC 5	0.37 ± 0.046	0.3344 ± 0.0341	1.106	0.631
BTC 6	1.3172 ± 0.0477	1.061 ± 0.063	1.241	4.801
BTC 7	0.3265 ± 0.0227	0.3264 ± 0.0127	1.0003	27.829
BTC 8	0.2266 ± 0.0144	0.2124 ± 0.0103	1.067	38.018
EBTC1	46.8687 ± 0.851	1.8563 ± 0.0918	25.248	1.116
EBTC2	30.4593 ± 1.0404	3.0844 ± 0.1191	9.875	15.461
EBTC3	0.8626 ± 0.0649	0.1188 ± 0.01443	7.26	0.759
EBTC4	531.9172 ± 2.340	0.636 ± 0.094	836.35	1.017
EBTC5	46.303 ± 2.827	4.014 ± 0.2672	11.535	2.514
EBTC6	1602.692 ± 24.59	36.8641 ± 0.317	43.476	12.341
EBTC7	3.4765 ± 0.2576	0.3142 ± 0.1046	11.065	0.9767
EBTC8	0.8797 ± 0.0627	0.1203 ± 0.035	7.313	0.774

Table 6.4 Operations by RDF3X for the BTC queries

Query	Number of merge joins	Number of hash joins	DISTINCT
BTC 1	3	0	
BTC 2	3	0	
BTC 3	4	0	
BTC 4	5	1	
BTC 5	2	1	√
BTC 6	2	2	√
BTC 7	4	3	√
BTC 8	3	1	
EBTC 1	1	1	
EBTC 2	2	1	
EBTC 3	1	1	
EBTC 4	1	1	
EBTC 5	2	1	
EBTC 6	4	1	
EBTC 7	2	1	√
EBTC 8	1	1	√

Table 6.4 presents the query operations performed by the original RDF3X approach. The additional BTC queries, which we have used in the experiments, are provided below as follows:

We assume that all BTC queries define the namespaces `rdf`, `rdfs`, `foaf`, `dbpedia`, `purl`, `sioc`, `skos`, `atom`, `geoont`, `geocountry`, and `geopos` by

```

PREFIX rdf:
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:
    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia: <http://dbpedia.org/property/>
PREFIX purl:    <http://purl.org/dc/elements/1.1/>
PREFIX sioc:    <http://rdfs.org/sioc/ns#>
PREFIX skos:    <http://www.w3.org/2004/02/skos/core#>
PREFIX atom:    <http://www.w3.org/2005/>
PREFIX geoont:  <http://www.geonames.org/ontology#>
PREFIX gecountry:
    <http://www.geonames.org/countries/#>
PREFIX geopos:
    <http://www.w3.org/2003/01/geo/wgs84_pos#>.

```

Query EBTC 1:

```

select * where {
?x foaf:depiction ?d;
    dbpedia:hasPhotoCollection ?y.
?y foaf:maker ?m.}

```

Query EBTC 2:

```

select * where {
?x purl:title "Wimbledon_College_of_Art";
    sioc:has_creator ?c; sioc:links_to ?l.
?l purl:title ?t.}

```

Query EBTC 3:

```

select * where {
?x skos:subject <http://dbpedia.org/resource/
    Category:1960_in_Formula_One>;
    dbpedia:wikilink ?l.
?l foaf:name ?n.}

```

Query EBTC 4:

```

select * where {
?m purl:title ?t.
?x foaf:made ?m; foaf:nick ?i.}

```

Query EBTC 5:

```

select * where {
?x atom:Atomuri ?u; atom:Atomname ?n.
?y atom:Atomname ?n; atom:Atomemail ?m.}

```


Query EBTC 6:

```
select * where {
?x geoont:name> ?n1; geopos:lat ?l;
    geoont:inCountry geocountry:DE.
?y geoont:name ?n2; geopos:lat ?l;
    geoont:inCountry geocountry:DE.}
```

Query EBTC 7:

```
select distinct ?t where {
?x purl:title "Wimbledon_College_of_Art";
    sioc:has_creator ?c; sioc:links_to ?l.
?l purl:title ?t.}
```

Query EBTC 8:

```
select distinct ?nwhere {
?x skos:subject <http://dbpedia.org/resource/
                                Category:1960_in_Formula_One>;
    dbpedia:wikilink ?l.
?l foaf:name ?n.}
```

6.11.2.3 Performance Gains

Several main factors contribute to the significant performance gains from our presorting numbering scheme, when processing queries with larger intermediate results:

1. A merge join with a fast sorting phase is more efficient than a hash join.
2. It often occurs that one operand of joins is already sorted accordingly. Under such cases, RDF3X-Sort only needs to sort another operand. In comparison, when using a hash join, two operands must be processed anyway.
3. Using the SIP strategy merge joins can benefit more than hash joins, by jumping over much larger gaps.

6.12 Summary and Conclusions

We develop a new and efficient approach to computing joins in memory by dynamically restricting triple patterns and using seven indices.

Our experimental evaluation shows that our proposed approach for joining the result of triple patterns in memory is faster than in-memory variants of disk-based join algorithms and common in-memory SPARQL database engines. Concretely, the average execution of all SP²B benchmark queries (Schmidt et al. 2009) of our approach is at least two times faster (see Fig. 6.25) than the compared approaches and common in-memory SPARQL database engines.

For efficiently querying the large-scale Semantic Web, we propose a sorting numbering scheme in order to fast sort solutions of SPARQL queries. Having the fast sorting capability, a merge sort join can be efficiently applied to compute the joins, data of which are unsorted. For large data sets, in combination with SIP strategies, the application of merge joins instead of hash joins leads to remarkable performance improvements. Elimination of duplicates also benefits significantly from the fast sorting capability. Our approach neither requires more space in the indices nor has extra update costs, since we use the ids of RDF terms as presorting numbers. By using histograms for the determination of the subranges for the buckets to be sorted in external mass storage, we ensure a good distribution between the buckets even if there are gaps in the values to be sorted, that is, intervals in the ids, which do not occur in the values, are quite common in large datasets, and could occur after updates.

Our experimental results show that merge joins using our fast sorting algorithm are more efficient than hash joins, and our fast sorting capability is a big benefit for elimination of duplicates. Our sorting numbering scheme significantly speeds up querying very large Semantic Web databases.

Chapter 7

Streams

Abstract Data streams are becoming an important concept and used in more and more applications. Processing of data streams needs a streaming engine. The streaming engine can start query processing once initial data is available. This capability is especially important for real-time computation and for long-relay transmission of data streams. In this chapter, we introduce stream processing by a demonstration of a monitoring system of eBay auctions, which is based on our RDF stream engine and can analyze eBay auctions in a flexible way. Using our monitoring system, users can easily monitor the eBay auctions information of interest, analyze the behavior of buyers and sellers, predict the tendency of auctions, and make more favorable decisions.

7.1 Introduction

A growing number of applications in areas such as network monitoring, sensor networks, and auction industry are using continuous *data streams* rather than finite stored datasets. Processing and querying data streams require long-running *continuous queries* as opposed to one-time queries.

Data produced over time form data streams. Data streams having no end are called *infinite data streams*. (Infinite) data streams are generated from, for example, sensors, which constantly obtain data from their environment. In order to determine useful conclusions (like a probably upcoming earth quake) from data streams, we need to consider the infinite nature and support the computation of intermediate results based on a *window*, which contains the recent data of the infinite data stream. In many scenarios, the intermediate results must be calculated in a timely fashion; for example, a probably upcoming earthquake must be detected as early as possible allowing no delays for the computations.

Streaming query engines operating on data streams can (a) discard irrelevant input as early as possible, and thus save processing costs and space costs, (b) build indices only on those parts of the data, which are needed for the evaluation of the query, and (c) determine partial results of a query earlier, and thus evaluate queries more efficiently.

Stream-based processing enables more efficient evaluation not only in local scenarios, where the data are stored and the query engines run on the same computer, but also in many other applications, for example,

- In integrating data over networks such as the Internet, in particular from slow sources. It is desirable to progressively process the input before all the data are retrieved
- In continuous query processing over infinite data streams (e.g., Arasu et al. 2006), generated by, for example, sensors. *Continuous query processing* (e.g., Arasu et al. 2006) evaluates queries periodically.
- In selective dissemination of information, where RDF data have to be filtered according to requirements given in a query
- In pipelined processing, where data are sent through a chain of processors, and the input of each processor is the output of the preceding processor.

The data format of the Semantic Web is RDF, and a large amount of data are described using RDF. SPARQL (Prud'hommeaux and Seaborne 2008) is the standard RDF querying language and has been extended by several contributions to support operations in infinite RDF data streams (see e.g., Bolles et al. 2008; Barbieri et al. 2009). In this chapter, we demonstrate our streaming query engine by a real-world case: monitoring the eBay auctions based on querying a real-time eBay RDF data stream. Our demonstration application is available online and can be downloaded from (Fell et al. 2010). We will explain special operators for stream processing afterward in detail.

7.2 eBay

eBay (eBay 2010a) is a popular online auction and shopping website. Through eBay individuals and business sell and buy a wide variety of goods and services, and millions of items are auctioned daily. Furthermore, the eBay Developers Program (eBay 2010b) offers several eBay web services, with which new applications, tools, and value-added services can be created in order to meet the diverse needs of buyers and sellers on eBay.

The eBay web services use domains and aspects describe the auctioned items. A domain represents a kind of items, the aspects describe the characteristics of items in a given domain, and items are instances of a domain. For example, the book can be a domain, and the title, author, pages, and price can be the aspects of the book domain. A book entitled “Stream Processing” auctioned in eBay is an instance of the book domain. The information model¹ used by eBay is very similar to the RDF data model.

¹<http://developer.ebay.com/DevZone/finding/Concepts/FindingAPIGuide.html>

Therefore, we can use RDF to describe eBay auctions, thus leveraging SPARQL and RDF tools to query and process auction data. While the eBay’s Finding API supports the functionality of searching and browsing items listed on eBay, the RDF query language SPARQL provides more powerful capabilities than the eBay Finding API.

7.3 Monitoring eBay Auctions

By monitoring the real-time eBay auctions of interest, the buyers and sellers can predict auction tendencies and make better decisions. Furthermore, it also helps the economists and researchers in analyzing various aspects of buying and selling behavior. In this section, we demonstrate how the users of our system can easily query and monitor the eBay auction information in which they are interested.

7.3.1 Monitoring System

Figure 7.1 describes our system of monitoring eBay Auctions. Our stream generator interacts with the eBay platform using the eBay web services. In this figure, the stream generator calls the eBay server with the function `findItemsByKeywords` (“Wii”), which retrieves and returns the auction information matched by the keyword “Wii”. Once the first data element arrives, the stream generator transforms

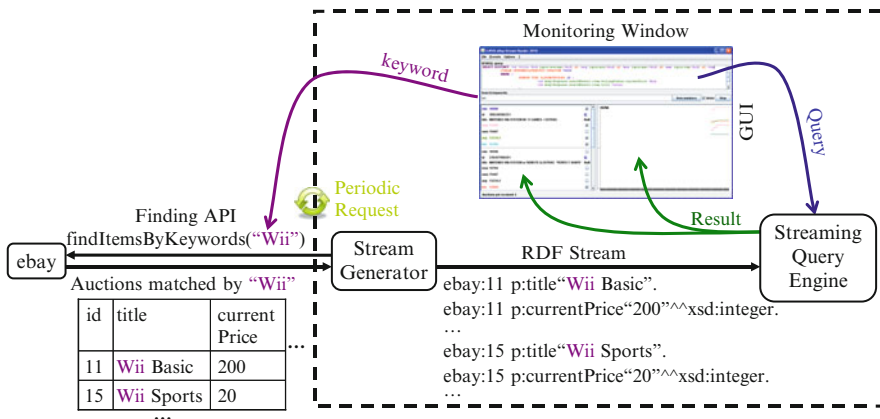


Fig. 7.1 Monitoring system of eBay auctions

it into the RDF format and sends it to the streaming query engine. The streaming query engine processes the SPARQL query on the RDF data stream and the results of the query are displayed in the monitoring window.

7.3.2 Demonstration

Figure 7.2 is a screen snapshot of the main window of our demonstration system. Users can specify the search keywords [see (1) of Fig. 7.2] or eBay item numbers or the webpage address of the eBay auction [see (2) of Fig. 7.2] for retrieving related information from eBay. The eBay item number can be found at its auction webpage. A SPARQL expression is used [see (3) of Fig. 7.2] to query the data returned by the eBay server. Several predefined SPARQL queries can be obtained by clicking the menu item *Presets* in the top menu.

After specifying the query information, click the button *Start* to start the communication with the eBay server, the generation of RDF data, and the evaluation of the SPARQL query over the RDF data stream. It might take some time to finish these processes, depending on various factors, for example, the speed of networks and the size of transmitted data.

The query result is displayed in the main window [see (4) of Fig. 7.2]. If a checkbox is marked in the query result, the numerical values are displayed in a chart [see (5) of Fig. 7.2]. The data are periodically retrieved and processed from eBay, and the query result is periodically updated. The old query result still remains in the charts when updated. Consequently, users can easily observe and monitor the changes over time.

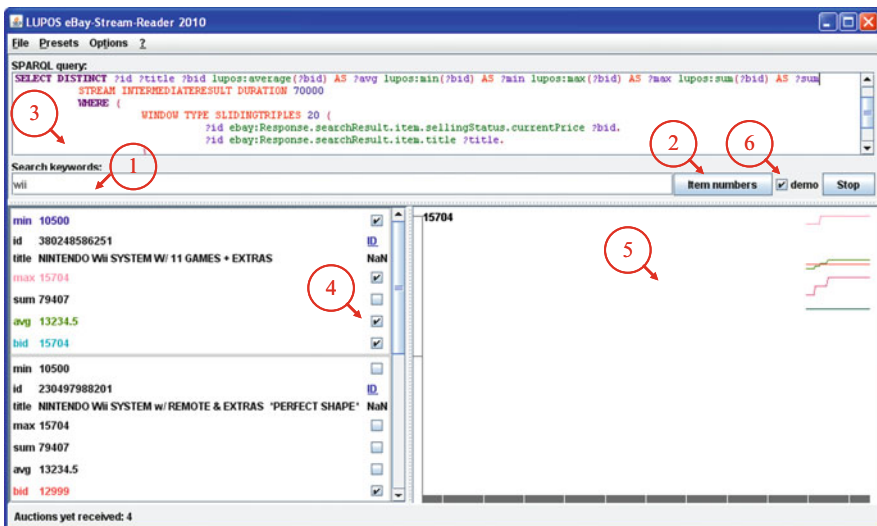


Fig. 7.2 Main window of our demonstration

7.3.3 Streaming SPARQL Engine

Our streaming SPARQL engine supports an extended version of SPARQL by allowing windows and the specification of the periods for updating the query result. Figure 7.3 describes such a query for the RDF stream. Line (5) specifies the query result to be updated every second, and lines (7–9) specify a window of the recent 100 triples of the RDF stream to be queried by the triple patterns in lines (8) and (9). Additional to SPARQL 1.0 (Prud’hommeaux and Seaborne 2008), we support aggregation functions [see lines (3) and (4)] such as average, min, max, and sum to determine the average, minimum, maximum, and summation.

Our demonstration also shows the internals of stream processing. Before processing a query, our streaming SPARQL engine parses the query and transforms it into a logically and physically optimized operator graph. If the checkbox “demo” (see (6) in Fig. 7.2 is enabled, a window of the evaluation demo will be popped after clicking the button *Start*. The window demonstrates single execution steps of the query processing, and displays the operator graph of the SPARQL query, and the information transmission between the operators [see (1) of Fig. 7.4].

The user can navigate through the processing steps by clicking on the next [see (2) of Fig. 7.4] or previous [see (3) of Fig. 7.4] step button. The user can also directly navigate to the first step [see (4) of Fig. 7.4] or watch an animation of the processing steps [see (5) of Fig. 7.4]. The processing of the RDF stream is initialized by sending a Start-Of-Evaluation-Message to each operator. Incoming triples are transmitted along the operator graph until a Triple Pattern operator, which is evaluated on the incoming triples. The result [see (1) of Fig. 7.4] is then transmitted to succeeding operators. The Window operator [see (6) of Fig. 7.4] handles the window of considered triples for query evaluation. If a triple is out of the window, then the Window operator transmits the information of deleting this triple downward. This information might cause a succeeding Triple Pattern operator to delete a certain solution. The information to delete a certain solution is transmitted to the succeeding operators, which have to delete this solution from their eventually used temporary indices and finally from the query result. The streaming engine triggers the periodic computation of the query by a Compute-Intermediate-Result-Message, and the query result is displayed [see (7) of Fig. 7.4].

```
(1) PREFIX ebay: <http://developer.ebay.com/DevZone/finding/CallRef/findItemsByKeywords.html#>
(2) PREFIX lupos: <http://www.luposdate.org/>
(3) SELECT DISTINCT ?id ?title ?bid lupos:average(?bid) AS ?avg lupos:min(?bid) AS ?min
(4)                lupos:max(?bid) AS ?max lupos:sum(?bid) AS ?sum
(5) STREAM INTERMEDIATERESULT DURATION 1000
(6) WHERE {
(7)   WINDOW TYPE SLIDINGTRIPLES 100 {
(8)     ?id ebay:Response.searchResult.item.sellingStatus.currentPrice ?bid.
(9)     ?id ebay:Response.searchResult.item.title ?title.}}
```

Fig. 7.3 Example query for RDF streams

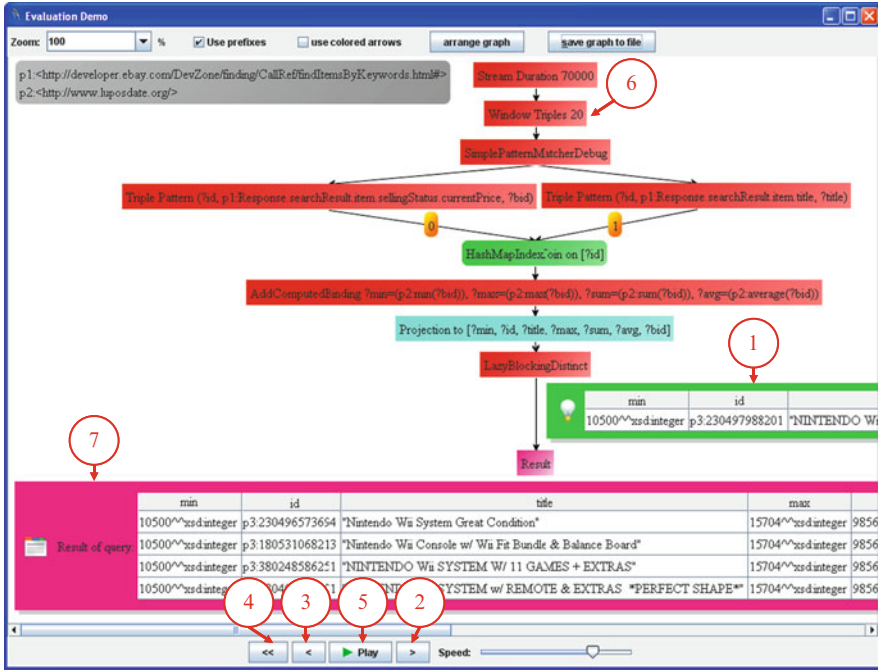


Fig. 7.4 Demonstration of evaluating RDF streams

Note that LUPODATE also supports the stepwise evaluation demo for the other SPARQL engines such as the main-memory engine and the RDF3X and Hexastore reimplementations, which do not process data streams.

7.4 Special Operators for Stream Processing

The SPARQL algebra must be extended by basically two types of operators for stream processing:

1. The first one called Stream operator triggers the periodic computation of query results. The Stream operator is the root in operator graphs of stream queries.
2. The second one called Window operator implements that query processing only considers certain recent data instead of all data.

7.4.1 Types of Stream Operators

Stream operators differ in the way they determine when they trigger succeeding operators to compute an intermediate query result:

- The Stream Triples operator triggers the computation of an intermediate query result after a given number of triples have been arrived (and processed) independent from the time of the last computation.
- The Stream Duration operator starts the computation of intermediate query results after a certain time is over independent from the number of arrived (and processed) triples.

Both operator types have their applications: While the Stream Duration operator guarantees up-to-date query results, the Stream Triples operator allows computing the query result only when there are many changes in the input.

7.4.2 *Types of Window Operators*

Window operators can differ how one can define the triples to be considered:

- The Window Triples operator considers only the last recent triples (up to a specified number of triples) for query processing. The Window Triples operator finds its applications, for example, whenever new triples update the values of older ones and these newer triples should be only considered during query processing.
- The Window Duration operator considers only those triples, which have been arrived in recent time (up to a specified time period), for query processing. The Window Duration operator is a necessity to compute aggregation functions over a certain time period, for example, the average temperature of the last hour.

More types of Window operators exist. We refer the interested reader to, for example, Arasu et al. (2006).

7.5 Related Work

We divide the related contributions into those dealing with data streams in general and those especially for Semantic Web streams:

7.5.1 *Data Streams in General*

The Chronicle (Jagadish et al. 1995) data model introduced data streams by describing chronicles as append-only ordered sequence of tuples and an algebra operating over chronicles as well as over traditional relations. Distributed stream management is supported in OpenCQ (Liu et al. 1999),
(continued)

NiagraCQ (Chen et al. 2000), and Aurora (Balakrishnan et al. 2004), which evolved into the Borealis project (Abadi et al. 2005). Babu and Widom (2001) address continuous queries over data streams, which evolved into the development of the CQL (Arasu et al. 2003, 2006; Munagala et al. 2007) query language tailored for data streams. Law et al. (2004, 2005), and Bai et al. (2006) deal with mining data streams. Especially, (Bai et al. 2006) extensively consider data aggregation in streams. Rewriting techniques for streaming aggregation queries are discussed in Golab et al. (2008).

7.5.2 *Semantic Web Data Streams*

We have proposed a SPARQL engine processing finite data streams in Groppe et al. (2007b) and have there defined corresponding logical and physical operators. To the best of our knowledge, our contribution in Groppe et al. (2007a, b) reports the first streaming SPARQL engine. At that time, we did not support window functions. Bolles et al. (2008) firstly introduced a window-based processing of RDF streams. Barbieri et al. (2009, 2010) have further extended the syntax of SPARQL by aggregates and timestamp functions, but restrict the functionalities by allowing only one Window per stream. Apart from supporting the aggregates and timestamp functions, we also allow several windows per stream. Walavalkar et al. (2008) describe a first approach to reasoning on data streams.

7.6 Summary and Conclusions

Our demonstration (Fell et al. 2010) shows the importance of streaming query engines in data stream applications. By using the RDF data stream and its querying language SPARQL, our monitoring system obtains big benefits in real-time data processing, and it provides users with the capabilities of observing, analyzing, and predicating the behavior and pattern of eBay buyers and sellers. Furthermore, our system can also stepwise display the internal processing steps of querying RDF streams, which helps one to better and easily understand the RDF stream processing technology.

Chapter 8

Parallel Databases

Abstract While a number of optimizing techniques have been developed to efficiently process increasing large Semantic Web databases, these optimization approaches have not fully leveraged the powerful computation capability of modern computers. Today's multicore computers promise an enormous performance boost by providing a parallel computing platform. Although the parallel relational database systems have been well built, parallel query computing in Semantic Web databases have not extensively been studied. In this work, we develop the parallel algorithms for join computations of SPARQL queries. Our performance study shows that the parallel computation of SPARQL queries significantly speeds up querying large Semantic Web databases.

8.1 Motivation

The Semantic Web databases are becoming increasingly large, and a number of approaches and techniques have been suggested for efficiently managing and querying large-scale Semantic Web databases. However, current algorithms are implemented for sequential computation and do not fully leverage the computing capabilities of current standard multicore computers. Therefore, the querying performance of the Semantic Web databases can be further improved by maximally employing the capabilities of multicore computers, that is, parallel processing of SPARQL queries.

A parallel database system improves performance through parallelization of various operations, such as building indexes and evaluating queries. While centralized database systems allow parallelism between transactions (multiuser synchronization), parallel database systems additionally use parallelism between queries inside a transaction, between the operators and within individual operators of a query.

Ideally, the speedup from parallelization would be linear – doubling the number of processing units should halve the runtime. However, very few parallel approaches can achieve such ideal speedup. Since the existence of nonparallelizable parts, most of them have a near-linear speedup for small numbers of processing

units, and the speedup does not become larger than a certain constant value for large numbers of processing units. The potential speedup of an algorithm on a parallel computing platform is governed by Amdahl's law (Amdahl 1967). The Amdahl's law discloses that a small portion of the problem, which cannot be parallelized, will limit the overall speedup available from parallelization by a constant value. For this reason, parallel computing is only useful for either small numbers of processors or highly parallelizable problems. We will see that SPARQL query evaluation is highly parallelizable.

Two major techniques of parallelism used in parallel database systems are *pipelined* parallelism and *partitioned* parallelism (DeWitt and Gray 1992). By streaming the output of one operator into the input of another operator, the two operators can work in series giving pipelined parallelism. N operators executed using pipelined parallelism can achieve a potential speedup of N . By partitioning the input data into multiple parts, an operator can be run in parallel using the multiple processing units, with each working on a part of the data. The partitioned data and execution provide partitioned parallelism.

Parallelism can be obtained from conventional sequential algorithms by using *split* and *merge* operators. The proven and efficient techniques (e.g. Mishra and Eich 1992) developed for centralized database systems can be leveraged in a parallel system enhanced with the split and merge operators. New issues that need to be addressed in parallel query processing include data partitioning and parallel join computation. The strategies of data partitioning contain (multidimensional) range partitioning, round-robin partitioning, and hash partitioning (Graefe 1993). A large number of optimizing techniques for parallel join processing of relational queries have been developed (e.g., Boral et al. 1990; DeWitt et al. 1986; Kitsuregawa et al. 1983; Kitsuregawa and Ogawa 1990; Schneider and DeWitt 1989, 1990; Wolf et al. 1990; Zeller and Gray 1990), most of which focus on parallel hash-join algorithms.

Parallel computing and parallel relational databases have been employed for many years, and a number of efficient techniques have been developed, which can be leveraged for parallel processing of SPARQL. However, optimizing techniques for parallel relational databases do not specialize on the triple model of RDF and triple patterns of SPARQL queries. In this chapter, we develop a parallel Semantic Web database engine based on the RDF- and SPARQL-specific properties. We focus on parallelization approaches for standard hardware with multiple processing cores and common memory and shared disks. This chapter contains contributions from (Groppe et al. 2011a).

The contributions of this chapter are as follows:

- Parallel join computations especially for
 - Hash joins using a distribution thread, and
 - Merge joins using partitioned input,
- An approach to computing operands in parallel, and
- An experimental evaluation, which shows the performance benefits of parallel data structures and algorithms in Semantic Web databases.

8.2 Types of Parallelisms

Different types of parallelisms can be used during query processing (see Fig. 8.1). We describe some of these types in detail in the following paragraphs:

1. A transaction contains several queries and updates of a database application. Transactions typically conform to the ACID properties of databases:

- (A) Atomicity: A transaction should be processed atomic; that is, the effects of all its queries and updates are visible to other transactions or none of them.
- (C) Consistency: The database should be left in a consistent state before and after the transaction.
- (I) Isolation: The transaction should be processed isolated; that is, the effect of the transaction should be the same as when all transactions are sequentially processed.
- (D) Durability: The effect of a successful transaction must be durable even after system crashes, damages of storage systems, or other erroneous software or hardware.

The multiuser synchronization of databases ensures the ACID properties for transactions, but allows processing different transactions in parallel (as much as possible considering conflicts between different transactions).

2. As a transaction contains several queries and updates, these queries and updates can be processed in parallel if they do not depend on each other: If an update

1. Inter-Transaction-Parallelism (Multi-User Synchronisation)



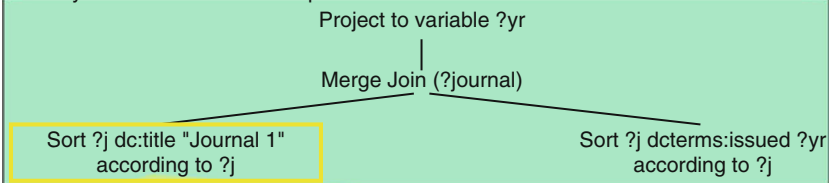
2. Intra-Transaction-Parallelism and Inter-Query-Parallelism

```

Start Transaction;
Exec sparql ... SELECT ?a WHERE {
  ?a rdf:type bench:Article; swrc:pages ?v. };
Exec sparql ... INSERT { dc:journal1 rdf:type bench:Article. };
Exec sparql ... SELECT ?yr WHERE {
  ?j dc:title "Journal 1. ?j dcterms:issued ?yr.};
Commit Transaction;

```

3. Intra-Query-Parallelism and Inter-Operator-Parallelism



4. Intra-Operator-Parallelism

```

Parallel Sorting of ?j dc:title "Journal 1" according to the variable ?j

```

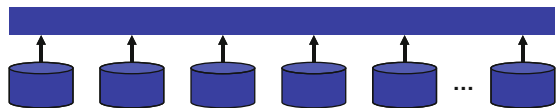
Fig. 8.1 Types of parallelisms in query processing

inserts, deletes, or modifies a triple, which influences the result of another query in the transaction, then the update and query must be processed in the order they occur in the transaction. The transaction in Fig. 8.1 contains an insertion of the triple *dc:journall rdf:type bench:Article*, which is matched by the triple pattern *?a rdf:type bench:Article* of the first query, such that the first query must be processed before the insertion. However, the last query does not contain any triple pattern matching the triple *dc:journall rdf:type bench:Article*, and therefore the last query can be processed in parallel to the insertion and also in parallel to the first query of the transaction.

3. A query (and also an update) of a transaction is transformed into an operator-graph consisting of several operators to be executed by the database system. Two forms of parallelisms can be applied here: The first form applies operators independent from each other in parallel. The second uses pipelining, where already computed intermediate results are transferred to subsequent operators as early as possible, which already processes this intermediate result further leading to a smaller memory footprint and to saving i/o accesses. We have already described pipelining in the chapter about physical optimization.
4. Many operators themselves can use parallel algorithms to determine their results. Prominent examples of such operators are sorting and join operators. For data parallelism, one tries to distribute the data into different disjoint fragments, such that operations like sorting or joins can be done in parallel in the different fragments. The extent of parallelism can be chosen dependent on the size of data; that is, larger data can be distributed into more fragments, such that more computers can be used to process the data in parallel leading to scalable solutions. Furthermore, data parallelism can be combined with pipelining. There are two forms of I/O parallelism (see Fig. 8.2):

- The access parallelism uses different I/O devices like hard disks to process one job. Access parallelism is important for data parallelism and the distributed fragments should be therefore stored on different I/O devices.
- During job parallelism independent jobs are processed on different I/O devices in parallel. Applying job parallelism is important for intertransaction parallelism and interquery parallelism.

(I) Intra-I/O-Parallelism (Access Parallelism)



(II) Inter-I/O-Parallelism (Job Parallelism)

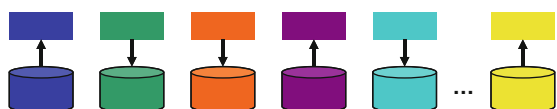


Fig. 8.2 Types of I/O parallelisms

8.3 Amdahl's Law

The batch speedup determines the response time improvement of parallel processing. The batch speedup for N computers is defined to be the response time when using one computer (and sequential algorithms) divided by the response time when using N computers, parallel algorithms, and the same data. Ideal would be a doubled batch speedup when using two times more computers (see Fig. 8.3). A linear improvement would be still fine, because one could determine the number of computers needed to obtain any response time one wants. However, experiments show that in typical cases the batch speedup does not increase or only slightly increases after a certain upper limit has been reached. In many cases, the batch speedup even decreases for more computers.

The reasons for the limits of scalability are startup and termination overhead of the parallel algorithms, inferences when accessing the logical and physical resources, overloads of individual computers, lock conflicts, limited partitioning possibilities of data, transactions and queries, and skew in the execution times of subtasks.

Amdahl's law now can determine an upper limit for the batch speedup, if the fraction of the execution time of the non-optimized, sequential part of the algorithm is known in relation to the overall execution time (see Fig. 8.4). Using this formula, one can determine a maximal batch speedup of 20 if the sequential fraction is only 5%.

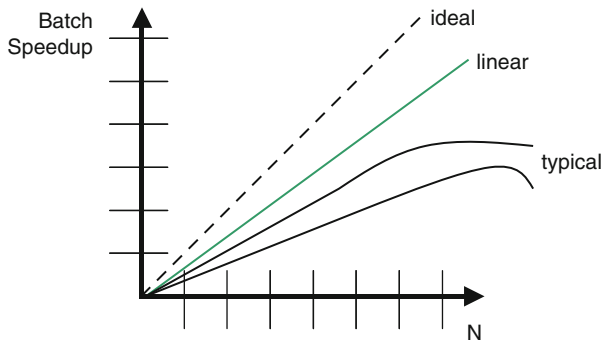


Fig. 8.3 Batch speedup dependent on the number N of computers

$$\text{Batch Speedup} = \frac{1}{(1-F_{\text{opt}}) + \frac{F_{\text{opt}}}{S_{\text{opt}}}}$$

F_{opt} = Fraction of the optimized (parallelized) component ($0 \leq F_{\text{opt}} \leq 1$)

S_{opt} = Speedup for the optimized (parallelized) component

Fig. 8.4 Formula for batch speedup

8.4 Parallel Monitors and Bounded Buffers

An important concept in parallel computing is parallel monitors. A parallel monitor is a high-level synchronization concept and is introduced in (Hoare 1974). It is a program module for concurrent programming with common storage, and has entry procedures, which are called by threads. The monitor guarantees mutual exclusion of calls of entry procedures: at most one thread executes an entry procedure of the monitor at any time. Condition variables may be defined in the monitor and used within entry procedures for condition synchronization.

An example of a parallel monitor is a bounded buffer. Mainly, a bounded buffer has two operations: *put* to store an element in the bounded buffer and *get* to retrieve an element from the bounded buffer. The bounded buffer has a specific constant limit for the number of stored elements. If a thread tries to put an element into a full bounded buffer, then the bounded buffer forces the calling thread to sleep until another thread calls *get* to retrieve one element. A *get* on an empty bounded buffer causes the calling thread to sleep until another thread puts one element inside.

The bounded buffers are typically used in producer/consumer patterns, where several threads produce elements and other threads consume these elements.

The advantage of bounded buffers in comparison to unbounded buffers is that the usage of the main memory is bounded, that is, consumer threads cannot store more elements than allowed by the main memory, thus avoiding the problem of out-of-memory errors. Therefore, we use bounded buffers for the communication between threads for the parallel join computation.

8.5 Parallel Join Using a Distribution Thread

When we use several threads for join computation (see Fig. 8.5) and the input is not partitioned yet, we have to partition the input data among these join threads using, for example, (multidimensional) range partitioning or hash partitioning.

The data must be partitioned according to the join variables in the following way: the solutions (of two join operands) with the same values bound to the join variables are distributed to the same thread. This ensures (a) that each join thread only involves the data in its own bounded buffers, and (b) that the overall join result is correct. Hash partitioning uses a hash function over the values of the join variables to determine to which join thread an input element is distributed. Hash partitioning promises good distributions and efficient partitioning computation. We hence use the hash partitioning for most parallel join processing.

For each operand of joins, we use a *data partitioning thread* to distribute the data into the bounded buffers of join threads (see Fig. 8.5). The join thread reads data from its two bounded buffers and performs a sequential join algorithm. If one join thread has finished its computation, then its result can be processed by succeeding operators (without waiting for the other join threads). This approach is the fastest one of parallelizing join algorithms such as the hash join and the index join.

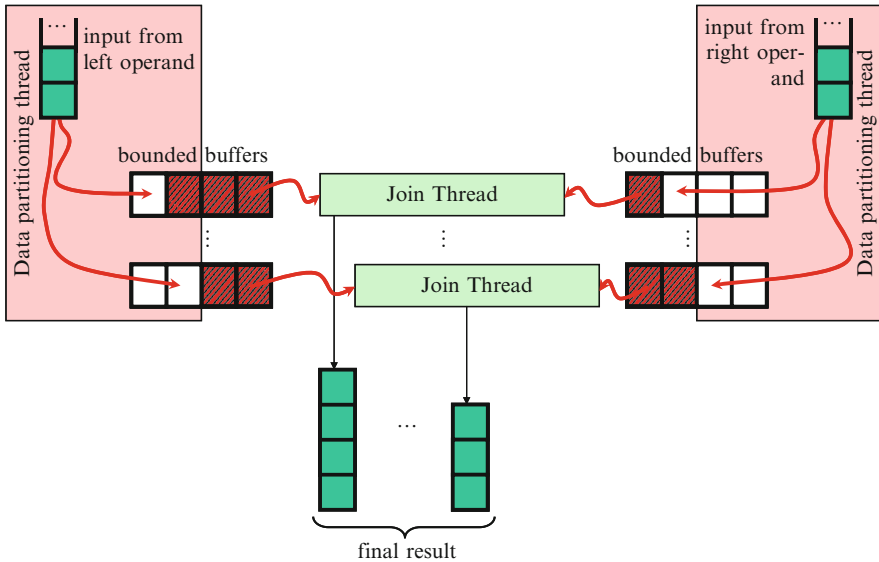


Fig. 8.5 Parallel join computation

When processing large data, the joins such as the disk-based hash join and index join involve complex operations during joining. Parallelizing these join algorithms, even plus the overhead of data partitioning, still can speed up join processing.

Merge joins on already sorted data are very fast and cheap in terms of I/O and CPU costs. The benefit from parallelizing merge joins cannot compensate the overhead of data partitioning even for large input data and large join results. We discuss how to parallelize merge joins in another way in the next section.

8.6 Parallel Merge Join Using Partitioned Input

As we mentioned before, a parallel merge join does not speed up the processing of joins if an additional partitioning process is needed. However, the processing performance benefits from the parallel computation of merge joins, if the input is already partitioned.

The merge join operator typically follows either the triple pattern operator in an operator graph, or the operators such as filters, which do not destroy the partitions of the input. Furthermore, the output of a merge join with such partitioned input is again partitioned, such that succeeding merge joins can use partitioned input as well. In order to generate a partitioned input for the merge join, we can retrieve the result of triple patterns using range partitioning (see Fig. 8.6). Range partitioning in comparison to hash partitioning has the advantage that the data in all the ranges can be read and processed in parallel.

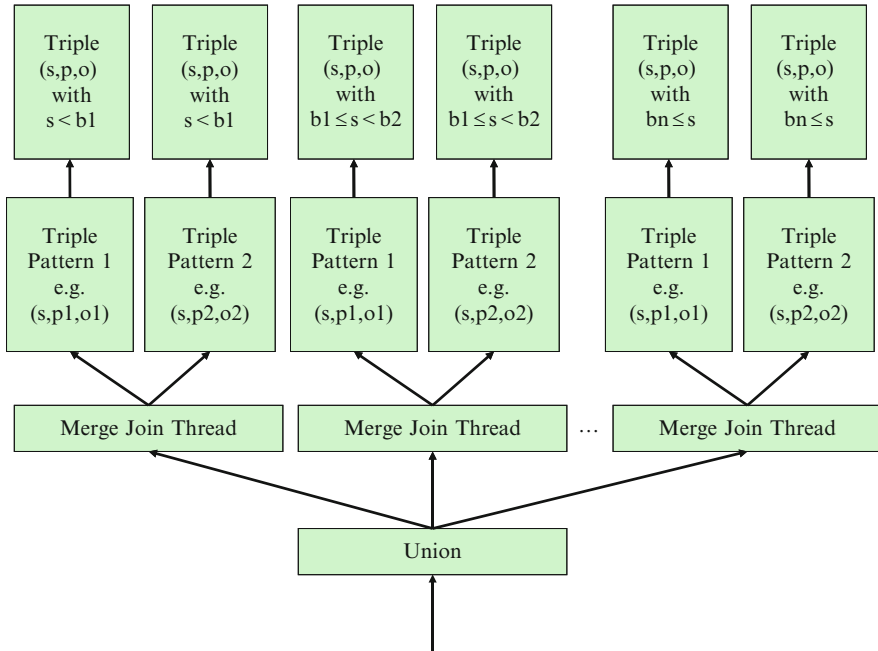


Fig. 8.6 Parallel merge join using range partitioning

SPARQL engines for large-scale data, such as *Hexastore* (Weiss et al. 2008) and *RDF3X* (Neumann and Weikum 2008, 2009), use six indices corresponding to six collation orders SPO, SOP, PSO, POS, OSP, and OPS to manage RDF triples. Depending on which positions in a triple pattern contain RDF terms (e.g., the subject and the object), one of the indices (e.g., SOP) is used to efficiently retrieve the data by using a prefix search. Using these collation orders, many joins can be computed using the fast merge join approach over sorted data. *RDF3X* employs just B^+ -trees for each collation order and prefix searches, thus gaining a simpler and faster index structure than *Hexastore*.

Employing B^+ -trees as the indices for each collation order has another advantage: The results of retrieving B^+ -trees can be very easily partitioned according to the range information in the histograms of triple patterns. For each kind of triple patterns, an equi-depth histogram (Piatetsky-Shapiro and Connell 1984) is constructed (see Chapter Logical Optimization) during the query optimization phase of our SPARQL engine. Among other information, each interval in this histogram contains the range and the number of triples allocated in this interval. We use special histogram indices to fast compute histograms over large datasets (see Chapter Physical Optimization).

Figure 8.7 describes how to get the partitioned results of a triple pattern using range partitioning: We assume that the data in the ranges $[(3, 2, 1), (3, 4, 7)]$ will be distributed to the first partition and the data in $[(3, 5, 5), (3, 7, 4)]$ to the second

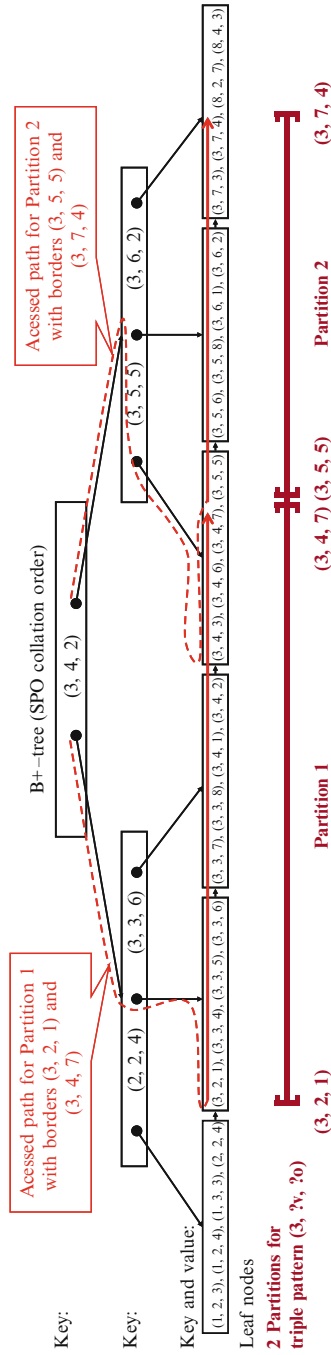


Fig. 8.7 B⁺-tree accesses for partitioned input. Integer ids are used instead of RDF terms as components of the indexed triples

partition. Two partitioning threads can perform the range partitioning in parallel: One first searches for the border (3, 2, 1) in the B^+ -tree and then follows the chain of leaves until the border (3, 4, 7), and the retrieved data belong to the first partition; another starts searching from the border (3, 5, 5) until the border (3, 7, 3) and retrieves the data for the second partition.

All approaches described so far for parallel joins apply also for the computation of OPTIONAL constructs. They are left outer-joins and can hence be analogously parallelized.

8.7 Parallel Computation of Operands

Another way to parallelize joins is to process their operands in parallel. In order to parallelize the processing of join operands, we use two *operand threads* (see Fig. 8.8). An operand thread computes its operand and puts the result into its bounded buffer.

Join approaches such as hash joins first read in the whole data of one operand and afterward start reading from the other operand. For such joins, the parallelism is not high, depending on the size of the bounded buffer. For the joins such as merge joins, which synchronously process the operands' data, two operand threads can work in parallel.

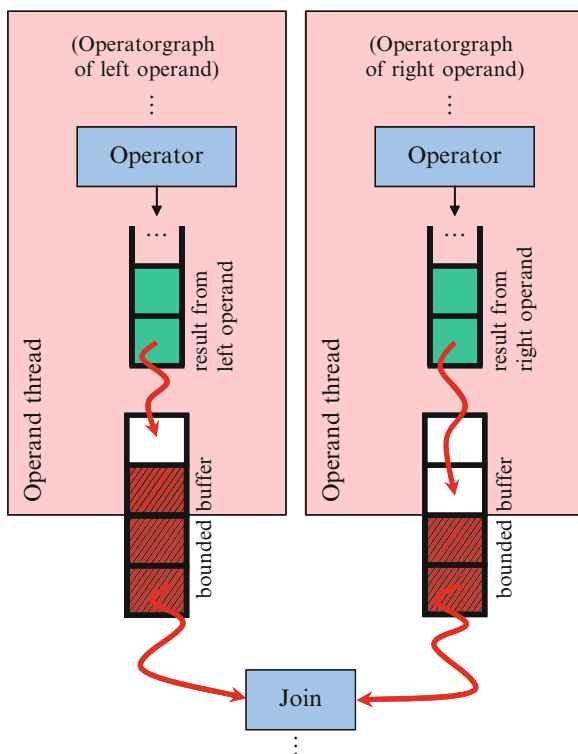


Fig. 8.8 Parallel computation of operands

However, advanced techniques such as sideways information passing (SIP) (Neumann and Weikum 2009) cannot be applied to parallel computation of operands, as using SIP to compute the next solution of one operand relies on the current solution of the other operand.

The experiments show that the parallel computation of operands speeds up the evaluation of some queries, especially if the processing of operands involves complex computations, but the previously discussed approaches for parallel joins are superior.

8.8 Performance Evaluation

We study the performance benefits for parallel join computations. For these experiments, we focus on the index approach *RDF3X* in (Neumann and Weikum 2008, 2009), since it is similar to Hexastore in (Weiss et al. 2008), but uses a simpler and faster index structure than (Weiss et al. 2008). We compare the pure *RDF3X* approach, that is, using sequential join algorithms, with several parallel versions of the *RDF3X* approach: *PHJ RDF3X* is the *RDF3X* approach using a parallel hash-join algorithm with eight join threads and distribution threads; *PMJ RDF3X* uses parallel merge join algorithms with eight merge join threads with partitioned input; *PO RDF3X* computes the operands of the last hash join in parallel. Combinations such as *PMJ PO RDF3X* combine together several parallel approaches such as *PMJ RDF3X* and *PO RDF3X*.

The original *RDF3X* prototype (Neumann and Weikum 2008, 2009) has several limitations, as already explained in the chapter about physical optimization. In order to lift these limitations and avoid problems resulting from not supported features of the original *RDF3X* prototype, we use again our reimplementations of the *RDF3X* approach.

The test system for the performance analysis uses an Intel Core 2 Quad CPU Q9400 computers, each with 2.66 GHz, 4 GB main memory, Windows XP Professional (32 bit), and Java 1.6. We have run the experiments ten times and present the average execution times and the standard deviation of the sample.

We use the large-scale datasets of the Billion Triples Challenge (BTC) (Semantic web challenge 2009) and corresponding queries.

We have imported *all* over 830 million distinct triples of the Billion Triples Challenge. In comparison, the performance analysis in (Neumann and Weikum 2009) used only a subset of it.

The queries (BTC 1 to BTC 8) used by (Neumann and Weikum 2009) return very small intermediate and final results, which can be processed directly in memory. For these queries, the parallel approaches often do not show benefits and are dominated by their overhead. Therefore, we also use several additional queries (EBTC 1 to EBTC 8) with bigger cardinalities (see the Chapter Physical Optimization). Table 8.1 presents the processing times by the different approaches.

Although the queries BTC 1 to BTC 8 are designed to retrieve very small results and thus are favorable to the *RDF3X* using sequential algorithms, except for the query BTC 7, our parallel approaches have similar performance as the sequential

Table 8.1 Evaluation times (in seconds) for BTC Data. Entries in bold face part mark significantly fastest runtimes

Query	RDF3X	PHJ RDF3X	PMJ RDF3X	PH RDF3X	PMJ RDF3X	PH PMJ RDF3X	PO RDF3X	PO PHJ RDF3X	Size of results
BTC 1	0.047 ± 0.014	0.047 ± 0.026	0.047 ± 0.015	0.047 ± 0.033	0.045 ± 0.013	0.046 ± 0.028	0.046 ± 0.028	0.046 ± 0.028	2
BTC 2	0.042 ± 0.016	0.046 ± 0.01	0.119 ± 0.134	0.11 ± 0.13	0.12 ± 0.012	0.12 ± 0.01	0.12 ± 0.012	0.12 ± 0.01	48
BTC 3	0.452 ± 0.082	0.436 ± 0.085	0.656 ± 0.047	0.74 ± 0.05	0.454 ± 0.105	0.48 ± 0.04	0.454 ± 0.105	0.48 ± 0.04	34
BTC 4	44.9 ± 0.1	45.99 ± 0.13	31.98 ± 2.99	33.8 ± 0.9	46.1 ± 0.2	44.86 ± 0.12	46.1 ± 0.2	44.86 ± 0.12	3
BTC 5	3.58 ± 0.12	3.52 ± 0.04	3 ± 0.06	2.99 ± 0.1	5.2 ± 0.2	5.15 ± 0.13	5.2 ± 0.2	5.15 ± 0.13	0
BTC 6	1.61 ± 0.03	1.65 ± 0.08	0.959 ± 0.564	0.76 ± 0.04	0.63 ± 0.05	0.76 ± 0.1	0.63 ± 0.05	0.76 ± 0.1	1
BTC 7	0.629 ± 0.077	6.33 ± 0.07	6.96 ± 2.74	13.7 ± 0.7	170 ± 1	173 ± 1	170 ± 1	173 ± 1	0
BTC 8	0.381 ± 0.070	0.36 ± 0.05	0.88 ± 0.09	0.87 ± 0.07	0.52 ± 0.07	0.52 ± 0.08	0.52 ± 0.07	0.52 ± 0.08	0
EBTC1	153.2 ± 3.7	105.9 ± 1.7	153.16 ± 3.38	113.44 ± 3.08	149 ± 2	139 ± 2	149 ± 2	139 ± 2	798,553
EBTC2	6.05 ± 0.41	4.41 ± 0.71	6.58 ± 0.34	4.69 ± 0.14	6.58 ± 0.7	5.6 ± 0.9	6.58 ± 0.7	5.6 ± 0.9	1,206
EBTC3	2.08 ± 0.05	3.12 ± 0.072	2.13 ± 0.13	2.13 ± 0.17	1.97 ± 0.13	3.3 ± 0.12	1.97 ± 0.13	3.3 ± 0.12	57
EBTC4	1,883 ± 32	1,241 ± 26	1,844 ± 18	1,297 ± 21	1,834 ± 21	1,850 ± 37	1,834 ± 21	1,850 ± 37	134,588
EBTC5	136.9 ± 4.1	145 ± 20	134 ± 3	160 ± 29	129 ± 3	137 ± 18	129 ± 3	137 ± 18	3,856,586
EBTC6	2,810 ± 24	1,736 ± 85	2,649 ± 39	1,799 ± 60	2,677 ± 9	1,555 ± 38	2,677 ± 9	1,555 ± 38	58,849,326
EBTC7	6.4 ± 0.34	4.87 ± 0.85	6.8 ± 0.62	5.37 ± 0.61	7.4 ± 0.9	5.33 ± 0.84	7.4 ± 0.9	5.33 ± 0.84	18
EBTC8	2.15 ± 0.07	3.01 ± 0.07	2.10 ± 0.06	3.07 ± 0.012	2.06 ± 0.13	3.35 ± 0.16	2.06 ± 0.13	3.35 ± 0.16	57

one. Our parallel approaches significantly outperform the sequential approach for the queries BTC 4 (PMJ RDF3X being the fastest), BTC 5 (PH PMJ RDF3X being the fastest) and BTC 6 (PO RDF3X being the fastest). For the EBTC queries, PHJ RDF3X is the fastest approach when evaluating EBTC 1, EBTC2, EBTC4, and EBTC 7, PO RDF3X is the fastest approach for EBTC 5, and PO PHJ RDF3X is the fastest one for EBTC 6. Parallel join algorithms work well whenever join results are large: If a merge join has a large result, PMJ RDF3X or PH PMJ RDF3X belongs to the fastest ones. If a hash join has a large result, PHJ RDF3X is the fastest.

8.9 Performance Gains and Loss

Several main factors contribute to the gain and loss of performance from parallelizing Semantic Web database engines:

- (a) PHJ RDF3X performs best whenever hash joins have large results.
- (b) Whenever merge joins have to process large input and have large results, then PMJ RDF3X or PH PMJ RDF3X outperforms the other sequential and parallel approaches.
- (c) The overhead of parallel processing such as data partitioning will dominate if involved data are small in size.
- (d) Advanced techniques such as sideways information passing (SIP) (Neumann and Weikum 2009) cannot be applied to some parallel computations.

Therefore, it is the task of the query optimizer to estimate the sizes of the join results and choose the proper join algorithms.

8.10 Summary and Conclusions

Since the disappearing of single-core computers, parallel computing has become the dominant paradigm in computer architectures. Therefore, developing parallel programs to fully employ the computing capabilities of multicore computers is under the necessity. This is especially important for time-consuming processing like querying growingly large Semantic Web databases. In this work, we develop a parallel SPARQL engine, especially focusing on the parallelism of join computation. For different join algorithms, we propose different parallel processing in order to maximally gain from parallel computing.

Our experimental results show that parallel join computation outperforms sequential join processing for large join results; otherwise, the parallel overhead compensates the performance improvements through parallelization. Therefore, the query optimizer must decide when to apply parallel join approaches. The proper application of parallel computation can significantly speed up querying very large Semantic Web databases.

Chapter 9

Inference

Abstract Data contain given facts, which are explicitly expressed. If we have the facts that Nils is a child of Sven and Sven is a child of Josef, then we as humans know that Josef is the grandparent of Nils, which is also called *implicit* knowledge. However, machines cannot process implicit knowledge as humans can do. Machines must get to know how to transform implicit knowledge to explicit knowledge, that is, to facts, such that machine can process it. The transformation from implicit knowledge to explicit knowledge is often expressed by rules. The application of rules to determine new facts is called *inference*. Inference is a costly operation, often leading to higher costs than query processing. We propose different materialization strategies for inferred facts to optimize query processing on inferred facts in this chapter and examine their performance gains.

9.1 Introduction

The mechanism to derive new facts based on given ones according to rules is called *inference*. The rules can be user-defined, for example, in the case of RIF (Boley et al. 2009) rules, or being fixed for inference based on RDF Schema (Brickley and Guha 2004) or OWL (2) (Dean and Schreiber 2004; Motik et al. 2009) ontologies.

A rule typically contains a *rule head* (called *left side* or *consequent*) and a *body* (called *right side* or *antecedent*). If the antecedent of a rule is true, then the consequent of the rule can be derived as new fact. Many different notations for rules exist: For example, a rule with consequent $(c1, \text{subClassOf}, c3)$ and antecedent $(c1, \text{subClassOf}, c2)$ and $(c2, \text{subClassOf}, c3)$ is denoted by

$(c1, \text{subClassOf}, c3) :- (c1, \text{subClassOf}, c2), (c2, \text{subClassOf}, c3),$

by

IF $(c1, \text{subClassOf}, c2)$ AND $(c2, \text{subClassOf}, c3)$ THEN $(c1, \text{subClassOf}, c3),$

or by

$$\frac{(c1, \text{subClassOf}, c2)(c2, \text{subClassOf}, c3)}{(c1, \text{subClassOf}, c3)}$$

with $c1$, $c2$, and $c3$ being variables. This rule describes the transitive property of *subClassOf* relationships; that is, if the facts $(Jeep, subClassOf, Car)$ and $(Car, subClassOf, Vehicle)$ are given, we can derive the new fact $(Jeep, subClassOf, Vehicle)$ according to this rule. The rule given above is one of the inference rules for RDF Schema and OWL ontologies. Derived facts such as $(Jeep, subClassOf, Vehicle)$ must be considered during query processing; that is, a triple pattern $(?s, subClassOf, Vehicle)$ has also the solution $\{(?s = Jeep)\}$ based on the derived fact $(Jeep, subClassOf, Vehicle)$. Materializing, that is, precomputation, of the derived facts can speed up query processing, but may explode the size of administered data. Deriving for queries relevant facts for every query new does not have this disadvantage, but may slow down query processing significantly. In this chapter, we investigate how to optimize inference for query processing and propose several approaches each with its own advantages and disadvantages. In particular, we examine three different materialization strategies (1) No materialization of derived facts and deriving relevant facts for each query new, (2) optimized materialization of derived facts between facts of the ontology, and (3) optimized materialization of all derived facts. We focus on inference based on RDF Schema in order to demonstrate the main principles of the different materialization strategies. Note that RIF and OWL inference is based on more and more complex rules, but the main clues remain the same as for RDF Schema inference.

9.2 RDF Schema Inference Rules

There have been different sets of inference rules for RDF Schema proposed. First, the W3C itself proposes a set of inference rules in Hayes (2004). A simplified version of the inference rules in Hayes (2004) are proposed in Gutierrez et al. (2004). The authors in (Muñoz et al. 2007) claim that the inference rules in Hayes (2004) and in Gutierrez et al. (2004) are not sound and complete, and they propose their own ones being sound and complete.

We present here the simplified version of the W3C rules as proposed in Gutierrez et al. (2004). These rules give an overview and help to understand the main inference rules for RDF Schema. Muñoz et al. (2007) add some more rules (and let two minor rules [(VI) and (VII)] away), which all do not affect the core of RDF Schema inference. In later examples, especially in the operator graphs, we will use the official W3C inference rules (Hayes 2004). Note that our implementation can additionally use the inference rules of Gutierrez et al. (2004) and we yield similar results with the rule set of Gutierrez et al. (2004).

Triples $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ containing ontology information are those with $p \in \{domain, range, subPropertyOf, subClassOf\}$. The rules of Gutierrez et al. (2004), which require only triples containing ontology information are as follows:

$$(I): \frac{(c1, subClassOf, c2)(c2, subClassOf, c3)}{(c1, subClassOf, c3)}$$

$$(II): \frac{(p1, \text{subPropertyOf}, p2)(p2, \text{subPropertyOf}, p3)}{(p1, \text{subPropertyOf}, p3)}$$

The rules (I) and (II) generate the transitive closure of *subClassOf* and *subPropertyOf* relationships. The other rules are as follows:

$$(III): \frac{(p1, \text{domain}, c)(s, p1, o)}{(s, \text{type}, c)}$$

$$(IV): \frac{(p1, \text{range}, c)(s, p1, o)}{(o, \text{type}, c)}$$

$$(V): \frac{(s, \text{type}, \text{Property})}{(s, \text{subPropertyOf}, s)}$$

$$(VI): \frac{(s, \text{type}, \text{Class})}{(s, \text{subClassOf}, s)}$$

$$(VII): \frac{(p1, \text{subPropertyOf}, p2)(s, p1, o)}{(s, p2, o)}$$

$$(VIII): \frac{(c1, \text{subClassOf}, c2)(s, \text{type}, c1)}{(s, \text{type}, c2)}$$

Especially the rules (VII) and (VIII) determine instances and properties of super classes and super properties.

9.3 Materialization of Inference and Consequences for Query Optimization

If all derived facts are materialized (and thus are stored in the main indices), then queries can be processed as usual. For example, the query in Fig. 9.1 can be just processed as presented in the operator graph in Fig. 9.2. The operator graphs presented here are the ones of the stream engine, but analogous operator graphs exist for the other query engines.

```

PREFIX rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema#>

SELECT ?journal ?yr
WHERE {
  ?journal rdf:type foaf:Document.
  ?journal dc:title "Journal 1 (1940)"^^xsd:string.
  ?journal dcterms:issued ?yr. }

```

Fig. 9.1 Example SPARQL query for the different materialization strategies

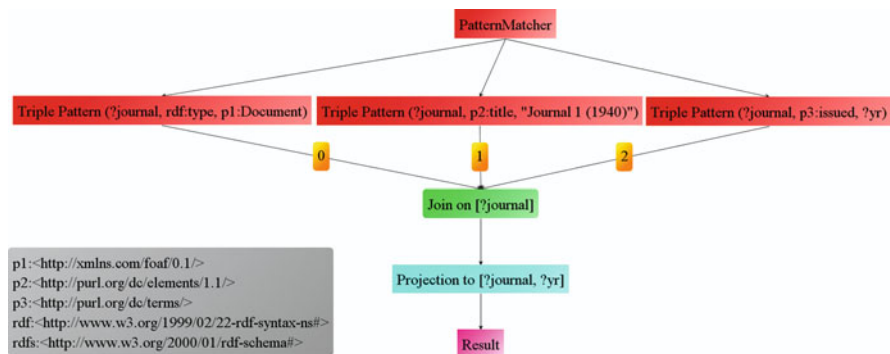


Fig. 9.2 Operator graph of the query in Fig. 9.1

However, if we materialize the inference requiring only triples containing ontology information [e.g., rules (I) and (II) for the RDF Schema rule set proposed in Gutierrez et al. (2004)], which we call *inference in the ontology layer*, then we can use this materialized inference to further optimize the operator graph: First, we have to generate the operator graph containing the query itself as well as the missing inference rules not for the inference in the ontology layer. For our example query in Fig. 9.1, we thus retrieve the operator graph in Fig. 9.3, where we consider the official inference rules of the W3C as proposed in Hayes (2004). Let us assume that the *subClassOf* relationships (*Journal*, *subClassOf*, *Document*) and (*Article*, *subClassOf*, *Document*) are given. Because of the materialization strategy, we know that all triples with predicate *subClassOf* are already inferred. Thus, because of the rule (VIII) of Gutierrez et al. (2004), we can replace the triple pattern (*?journal*, *rdf:type*, *Document*) with the union of the triple patterns (*?journal*, *rdf:type*, *Document*), (*?journal*, *rdf:type*, *Journal*), and (*?journal*, *rdf:type*, *Article*). Further logical optimizations lead to the very optimal operator graph as presented in Fig. 9.4 after 1,854 single transformation steps in our implementation. We present some of the logical optimization rules in later chapters. The materialization of the inference in the ontology layer can be expressed in the form of an operator graph (see Fig. 9.5) and afterward logically optimized (see Fig. 9.6). The optimized operator graph (see Fig. 9.6) only needs to be determined once and can be afterward directly used for any ontology.

If no inference has been materialized, then we have to add all inference rules in the operator graph of the query (see Fig. 9.7) and can afterward apply logical optimization (see Fig. 9.8).

9.4 Logical Optimization for Inference

We do not present all logical optimization rules here, as they are too many. However, we present a short overview of the goals and how they are reached in the logical optimization phase for inference. Therefore, we only describe the rules in an informal way such that the ideas behind them become clear.

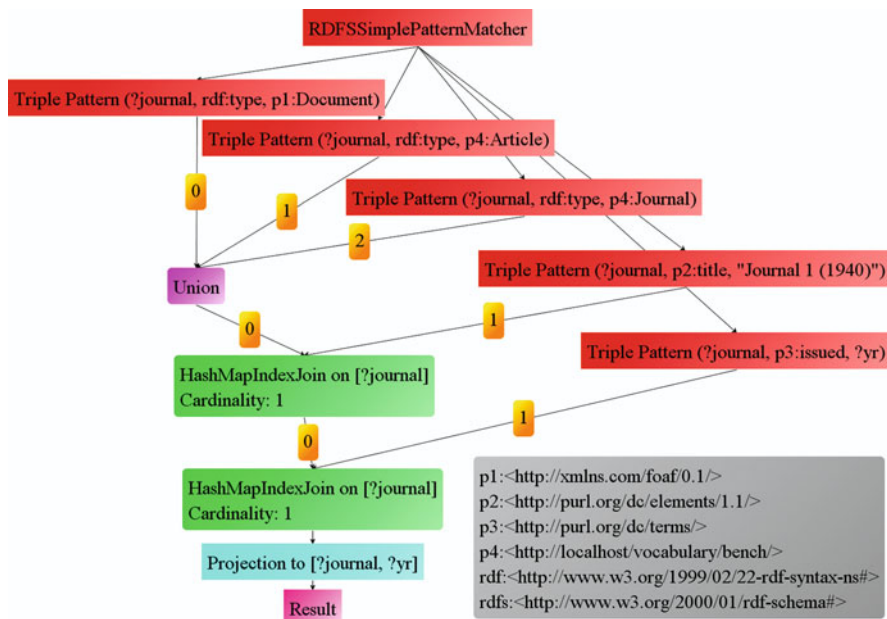


Fig. 9.4 Optimized operator graph of Fig. 9.3 after 1,854 single transformation steps

The logical optimization for inference without materialization or with materialization in the ontology layer has two main goals:

1. The first goal is to infer only that information which is necessary to answer a given query.
2. The second goal is to infer as late as possible during query processing in order to reduce the sizes of intermediate results.

New RDF triples are inferred by Generate operators in the initial execution plan. The first optimization rule determines which triple patterns could be matched by generated RDF triples from each Generate operator and connects the Generate operator with those triple patterns.

Generate operators generating RDF triples, which will never be matched by any triple pattern, can be already removed with its preceding operators. With this, we already fulfill the first goal and infer only that information which is necessary to answer a given query.

Now, we can determine the generated solutions of such a Generate – triple pattern combination, replace them with one special operator, and can try to move this special operator to inner subexpressions to infer as late as possible (goal two).

If we have materialized the inference within the ontology layer, we can try to already apply triple patterns matching RDF triples containing ontology information. The triple pattern is then replaced with the union of its solutions from applying it to the materialized ontology inference. Afterward, especially constant propagation

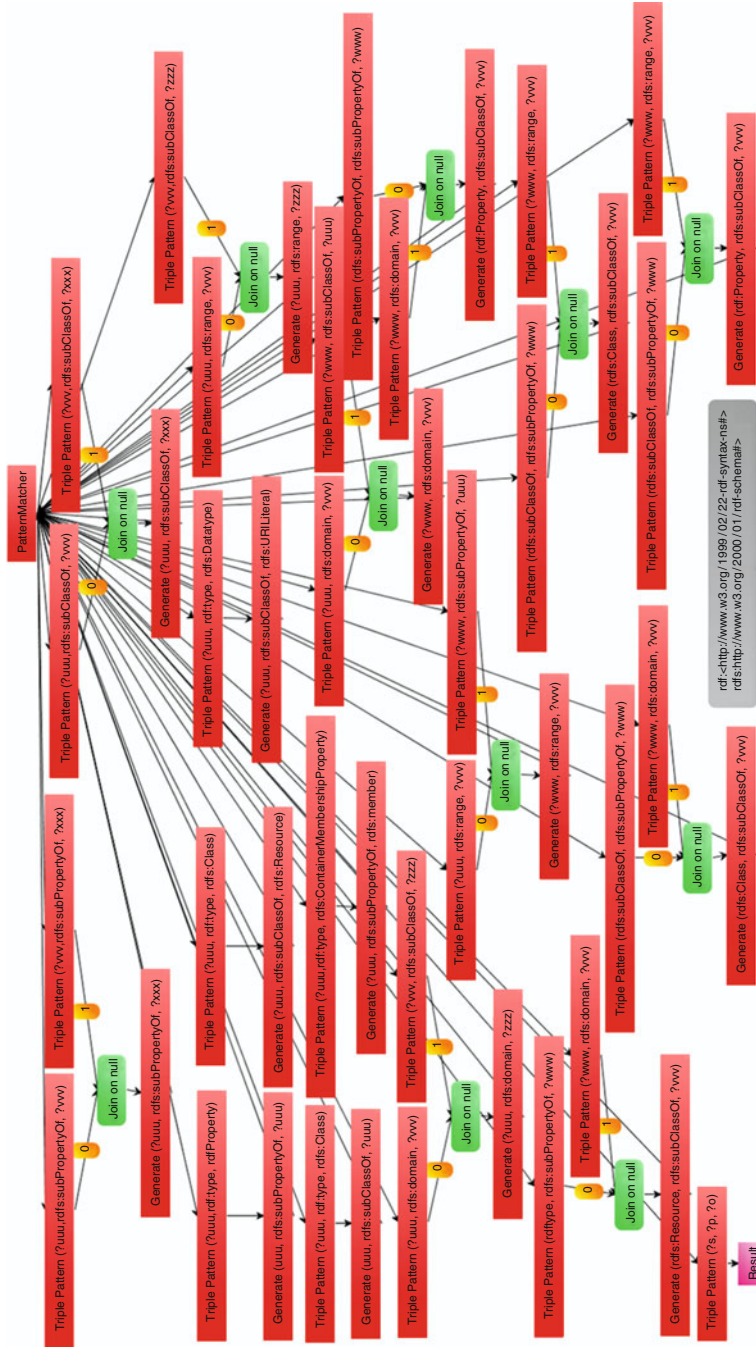


Fig. 9.5 Unoptimized operator graph for materializing the inference in the ontology layer

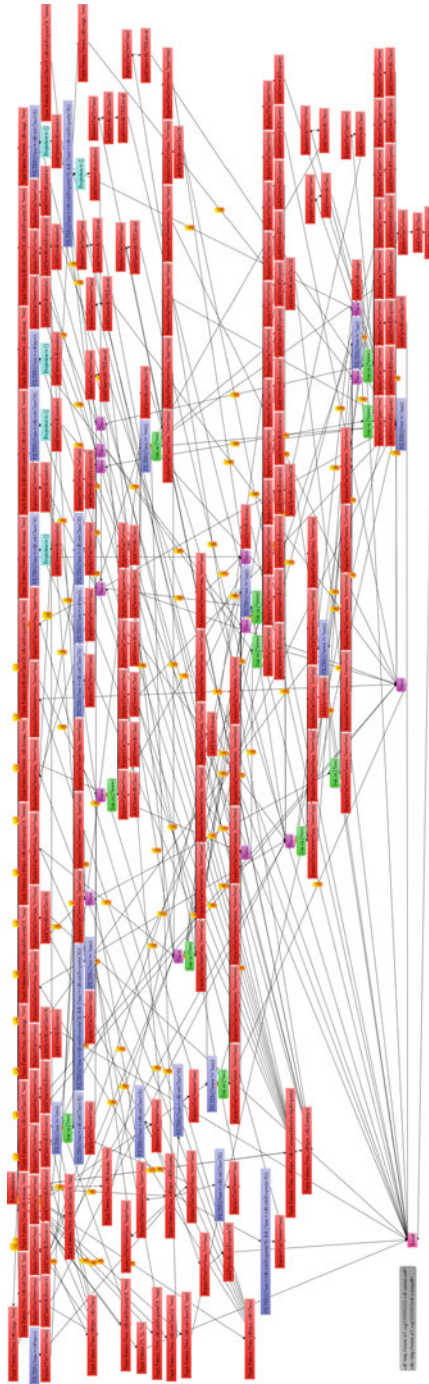


Fig. 9.6 Optimized operator graph of Fig. 9.5 after 144 single transformation steps

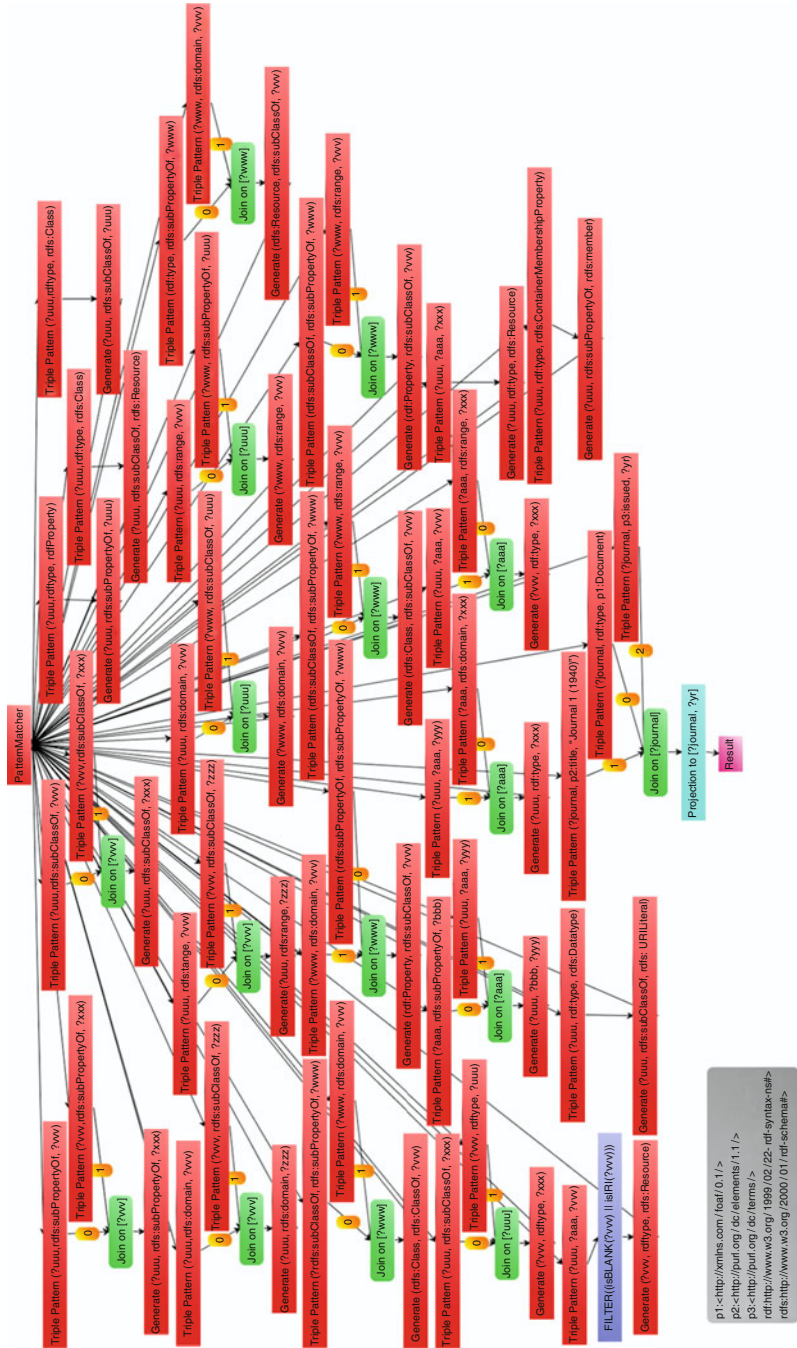


Fig. 9.7 Unoptimized operator graph of the query in Fig. 9.1 and all inference rules

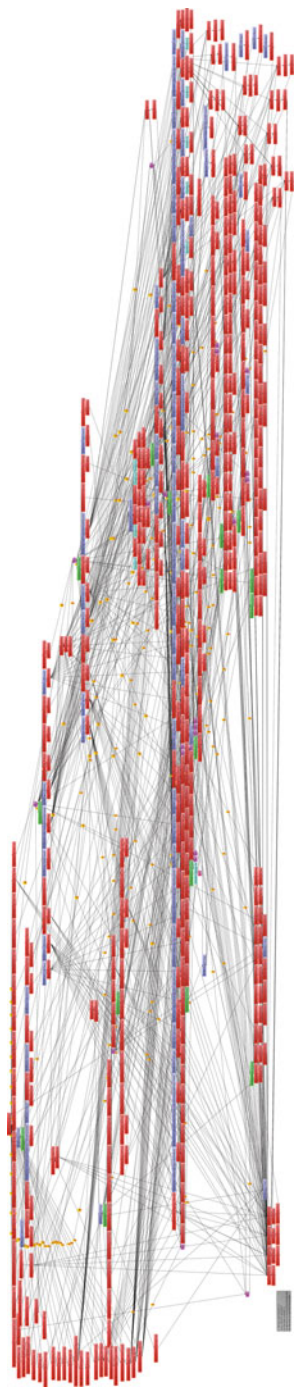


Fig. 9.8 Optimized operator graph of Fig. 9.7 after 344 single transformation steps

optimizes inference, such that we bind new variables based on the materialized ontology inference as late as possible.

9.5 Performance Analysis

The test system for the evaluation of queries uses a Dual Quad Core Intel CPU X5550 computer with 2.67 GHz, 6 GB main memory, Windows XP Professional (x64 Edition), and Java 1.6 64 bit. We have run the experiments ten times and present the average execution times as well as the standard deviation of the sample.

We have used the LUBM benchmark (Guo et al. 2005) for our experiments. The LUBM benchmark can generate data of different sizes and inference must be considered during query processing. The LUBM benchmark contains a university domain ontology. Although the original LUBM benchmark proposes an OWL ontology, we use an RDFS variant of the original LUBM benchmark for the considered RDFS inference. The used data contain 103,271 triples.

In our experiments, we use two different versions of rule sets for the inference. One rule set is the rule set of the W3C (Hayes 2004); the other is called Rudimentary RDFS and contains only the rudimentary rules (I) and (II) for inference in the ontology layer, and the rules (VII) and (VIII), which is the core of RDFS inference and is enough for most applications in practice. We have measured the times for materializing all inferable data, which we denote with *All*, for materializing no inferable data, which we denote with *None*, and for materializing the inferable data in the ontology layer, which we denote with *Ont*, in the tables of Figs. 9.9 and 9.10. We add *Rudimentary* to the name if Rudimentary RDFS is used instead of the W3C rule set for RDFS.

Not surprisingly, the index construction times (see Fig. 9.9) are the smallest for *None* and Rudimentary *None*, and the highest for *All*. Because of the small numbers of rules in the rule set for Rudimentary RDFS, the index construction time for Rudimentary *All* is relatively small, although it is the second highest one. The index construction times for Rudimentary *Ont* and *Ont* is only a little bit higher than those for (Rudimentary) *None*. This shows that most facts are inferred because of instance data and only few facts are inferred in the ontology layer.

Contrary to the index construction times, the query execution times (see Fig. 9.10) are the smallest for (Rudimentary) *All*, as all inferable facts are already materialized and no new facts need to be inferred during query processing. The query execution times for (Rudimentary) *None* are the highest, as necessary facts need to be inferred during query processing. The query execution times for (Rudimentary) *Ont* are between those of (Rudimentary) *All* and (Rudimentary) *None*, more often more close to (Rudimentary) *All* than to (Rudimentary) *None*.

The index only need to be constructed once and can be then used for many queries. Thus, (Rudimentary) *All* seems to be a good choice for high performance query processing. However, the space costs are high for (Rudimentary) *All*, as the index must hold all inferred facts. If there are frequent updates and each update

Rudimentary All	Rudimentary None	Rudimentary Ont	All	None	Ont
10.77 ±0.15	4.69 ±0.14	4.74 ±0.19	29.59 ±0.29	4.66 ±0.19	4.70 ±0.13

Fig. 9.9 Index construction times in seconds

Query	Rudimentary All	Rudimentary None	Rudimentary Ont	All	None	Ont
1	0.0079 ±0.0083	1.0766 ±0.3531	0.4359 ±0.1975	0.0047 ±0.0076	27.5422 ±2.6171	3.1063 ±0.0394
2	1.0282 ±0.0482	1.3813 ±0.3373	1.4016 ±0.1346	1.0985 ±0.2887	28.9484 ±2.2095	Out of Memory
3	0.011 ±0.0129	0.8202 ±0.0773	0.4624 ±0.2120	0.0047 ±0.0075	27.5984 ±1.7427	3.2579 ±0.1631
4	0.0406 ±0.0426	1.3704 ±0.4067	0.5625 ±0.2812	0.0234 ±0.0082	27.8609 ±1.4667	3.2703 ±0.1641
5	0.0141 ±0.0088	0.9797 ±0.2967	0.4422 ±0.1952	0.0095 ±0.0082	28.2812 ±1.5235	3.3765 ±0.1587
6	0.0046 ±0.0074	1.0203 ±0.3161	0.4407 ±0.1862	0.011 ±0.0107	26.8797 ±1.9840	3.1265 ±0.0881
7	0.1359 ±0.0824	Out of Memory	Out of Memory	0.1031 ±0.0286	Out of Memory	Out of Memory
8	0.3766 ±0.0903	1.9939 ±0.6878	107.7109 ±1.6675	0.3188 ±0.0255	28.7234 ±1.8121	110.5234 ±1.3026
9	1.8031 ±0.2309	Out of Memory	Out of Memory	1.7406 ±0.0109	Out of Memory	Out of Memory
10	0.0125 ±0.0124	1.0516 ±0.3734	0.4202 ±0.1731	0.0031 ±0.0065	28.2062 ±3.7902	3.211 ±0.1180
11	0.0015 ±0.0047	0.9406 ±0.3032	0.4374 ±0.1972	0.0016 ±0.0051	26.7531 ±1.9667	3.089 ±0.0584
12	0.0063 ±0.0077	1.0452 ±0.2812	0.4422 ±0.1821	0.0079 ±0.0083	28.1656 ±1.3778	3.1125 ±0.0610
13	0.0094 ±0.0081	1.0968 ±0.3453	0.4891 ±0.236	0.0063 ±0.0081	27.611 ±1.6638	3.3015 ±0.1704
14	0.0078 ±0.0082	0.9593 ±0.3314	0.4422 ±0.1880	0.0329 ±0.0090	26.7295 ±2.1833	3.1047 ±0.1027

Fig. 9.10 Query execution times in seconds

requires materializing all inferable facts again, then this variant has too high costs. For very frequent updates, the variant (Rudimentary) None seems to be a reasonable choice, as query processing anyway needs to determine all relevant inferable facts for each query again, as the data already changed most probably since the last query execution. (Rudimentary) Ont seems to be good compromise between both

extreme cases: Having moderate space costs as well as update costs and a high query processing performance.

9.6 Related Work

OWL reasoners are used to validate RDF data and to infer new facts based on OWL ontologies.

Table-based provers for description logic such as Pellet (Sirin et al. 2006), RacerPro (Haarslev and Möller 2003), or FaCT++ (Tsarkov and Horrocks 2006) are an obvious choice as OWL reasoners because of their traditional roots in inference calculus. These provers support a big subset of OWL, and many optimization techniques for increasing their performance have been developed (e.g., Fokoue et al. 2006).

An alternative is translating OWL ontologies into disjunctive Datalog programs for leveraging a Datalog engine. This approach is realized, for example, in KAON2 (Motik and Studer 2005). Many optimization techniques from deductive databases like magic sets can be applied, but the approach has performance problems, for example, when OWL ontologies restrict the cardinality of properties (Weithöner et al. 2007).

Other approaches such as OWLIM (Kiryakov et al. 2005) and OWL-JessKB (Kopena 2005) use standard rule engines to support OWL reasoning. However, these approaches are incomplete and consume much space because of their materialization strategy.

There exist some hybrid approaches such as QuOnto (Acciarri et al. 2005), Minerva (Zhou et al. 2006), Instance Store (Bechhofer et al. 2005), and LAS (Chen et al. 2005), which connect an external reasoner (often a table-based system) with a database. This allows handling big datasets, but the power of the used languages is very restricted.

9.7 Summary and Conclusions

Inference is necessary to process implicit knowledge. However, inference comes along with high costs. Optimizing inference is therefore a key issue for the success of Semantic Web technologies. We have investigated to (a) optimize a given query together with inference rules (and do not materialize any inferred facts), (b) to materialize all facts, and (c) to materialize facts within the ontology layer. We have focused on RDF Schema as basic ontology language. Variant (a) does not have any costs for materialization, but has the highest costs for query processing. Variant (b) has the lowest costs for query processing, but the highest for materializing inference. Variant (c) is a mixture of (a) and (b) and has moderate query processing as well as inference materialization costs.

Chapter 10

Visual Query Languages

Abstract The social web is becoming increasingly popular and important, because it creates the collective intelligence, which can produce more value than the sum of individuals. The social web uses the Semantic Web technology RDF to describe the social data in a machine-readable way. RDF query languages play certainly an important role in the social data analysis for extracting the collective intelligence. However, constructing such queries is not trivial because the social data are often quite large and assembled from a large number of different sources and because of the lack of structure information like ontologies. In order to solve these challenges, we develop a Visual Query System (VQS) for helping the analysts of social data and other semantic data to formulate such queries easily and exactly. In this VQS, we suggest a condensed data view, a browser-like query creation system for absolute beginners and a Visual Query Language (VQL) for beginners and experienced users. Using the browser-like query creation or the VQL, the analysts of social data and other semantic data can construct queries with no or little syntax knowledge; using the condensed view, they can determine easily what queries should be used. Furthermore, our system also supports a number of other functionalities, for example, precise suggestions to extend and refine existing queries. An online demonstration of our VQS is publicly available at <http://www.ifis.uni-luebeck.de/index.php?id=luposdate-demo>.

10.1 Motivation

The Social Web fosters collaboration and knowledge sharing and boots the collective intelligence of the society, organizations, and individual people. The significance of the Social Web is already beyond the field of computer science. The Semantic Web promises a machine-understandable web, and provides the capabilities for finding, sorting, and classifying web information, and for inferring new knowledge. The Semantic Web hence offers a promising and potential solution to mining and analyzing the social web.

Therefore, the emerging Social Semantic Web, the application of semantic web technologies to the social web, seems to be a certain evolution. In the Social

Semantic Web, the social data are described using the Semantic Web data format RDF, and the social data become hence machine-understandable.

Analysis of social data plays a critical role in, for example, extracting the collective intelligence, obtaining insights, making decisions, and solving problems. The Social Semantic Web gathers a huge amount of semantic data. For example, DBpedia (<http://wiki.dbpedia.org/About>) has extracted 479 million RDF triples from the social website Wikipedia (http://en.wikipedia.org/wiki/Main_Page). Analyzing such a large amount of data by browsing either the datasets or their visualizations is impractical. RDF query languages such as SPARQL (Prud'hommeaux and Seaborne 2008) are obviously an important tool for the analysis of large-scale social data.

Ontologies are used to describe the structure of the Semantic Web data. Due to the open world assumption of the Semantic Web, the data may contain structures, which are not described by the given ontologies. Ontologies are also often not given for considered data. This is especially true for the social data, which are contributed by a huge number of individual participants.

In order to formulate SPARQL queries for analyzing social data or other semantic data, the analysts have to hence look into the data as well as its ontology, besides having the knowledge of the SPARQL language. Looking into a large amount of social data is obviously inefficient and impractical. Therefore, in this work, we propose a condensed data view, which describes the structure of the initial large data, but uses a compact representation of the data.

We also develop a browser-like query creation system and a visual editor of SPARQL queries, which allows analysts to formulate SPARQL queries with little syntax knowledge of SPARQL. Using different frames for the browser-like creation and the visual editor within the same window, the user can use both approaches in parallel for query creation; that is, all effects during query editing are visible in the browser-like query creation as well as in the visual editor and the user can manipulate the query either in the browser-like query creation frame or in the visual editor. Furthermore, we support to extend and refine queries based on the concrete social data for constructing exacter queries for more precise and efficient data analysis during browser-like query creation as well as visual editing.

In order to demonstrate these approaches to facilitating the (social) data analysis, we develop a Visual Query System (VQS), which includes the following:

- Support of whole SPARQL 1.0.
- Browser-like query creation for absolute beginners.
- Visual editor of SPARQL queries, which hides SPARQL syntax from users and supports the creation of more complicated queries in comparison to browser-like query creation.
- Visual browsing and editing of RDF data.
- Condensed data view for avoiding browsing the initial large datasets.
- Extending and refining queries based on query results.
- Import and export of textual SPARQL queries.

10.2 Related Work

Query-by-form is used in many web applications for simple data access; users are provided a form, all fields of which are seen as parameters of a fixed query. Query-by-form is neither flexible nor expressive, as a form needs to be developed for each query.

Query-by-example (Zloof 1977) allows users to formulate their queries by selecting the names of the queried relations and fields and by entering keywords into a table.

As many databases are modeled conceptually using EER or ORM diagrams, a user can also query these databases starting from those diagrams by using *conceptual query languages*. Examples are EER-based (Czejdo et al. 1987; Parent and Spaccapietra 1989) and ORM-based (De Troyer et al. 1988; Hofstede et al. 1995) approaches. Ontology-aware VQSs (Russell and Smart 2008; Fadhil and Haarslev 2006; Vdovjak et al. 2003; Borsje and Embregts 2006; Catarci et al. 2004; OpenLink 2010) also belong to this class of visual query languages.

We classify the existing VQSs for the Semantic Web according to the support of suggestions for extending and refining queries. The VQSs, which provide suggestions for extending and refining queries, include vSPARQL/NITELIGHT (Russell and Smart 2008; Smart et al. 2008), GLOO (Fadhil and Haarslev 2006), EROS (Vdovjak et al. 2003), SPARQLViz (Borsje and Embregts 2006), SEWASIE (Catarci et al. 2004), and iSPARQL/OpenLink (OpenLink 2010). Other VQSs do not support suggestions, like RDF-GL/SPARQLinG (Hogenboom et al. 2010), MashQL (Jarrar and Dikaiakos 2008, 2009), and the one from DERI (Harth et al. 2006). All the contributions for extending and refining a query are based on ontologies. In comparison, our VQS LUPOSDATE-VEdit extends and refines a query based on concrete data.

The Semantic Web VQSs can be also classified according to the supported query languages. Some of the VQSs support SPARQL as query language (e.g., Borsje and Embregts 2006; Jarrar and Dikaiakos 2008, 2009; Hogenboom et al. 2010; Russell and Smart 2008; OpenLink 2010). The rest of the VQSs supports other query languages like triple pattern queries (Harth et al. 2006), RQL (Vdovjak et al. 2003), nRQL (Fadhil and Haarslev 2006), and conjunctive queries (Catarci et al. 2004). All the VQSs for SPARQL only support a smaller subset of SPARQL 1.0, whereas our LUPOSDATE VQS supports the whole SPARQL 1.0.

Other semantic web approaches use visual scripting languages, such as SPARQLMotion (SPARQLMotion 2008), Konduit (Möller et al. 2008a, b) and Deri Pipes (Tummarello et al. 2007). In these approaches, visual boxes and lines are used to connect different query and transformation steps, but the embedded queries must be inserted and edited in a textual form. This chapter contains contributions from (Groppe et al. 2011).

10.3 RDF Visual Editor

The core element of RDF data is the RDF triple $s p o$, where s is called the subject, p the predicate, and o the object of the RDF triple. A set of RDF triples builds a directed graph, called RDF graph. In an RDF graph, the subject and object are nodes and the predicate is a directed edge from the subject to the object. Therefore, the visual representation of RDF triples should conform to the RDF graph.

Table 10.1 contains textual and visual representations of two RDF triples. The visual representation is actually a screenshot of our RDF visual editor. In addition to browsing data, the RDF visual graphs can be used to update data easily. For example, the button $+$ is used to add a triple and $-$ to delete a triple easily.

10.4 SPARQL Visual Editor

The natural way to visualize the triple patterns of SPARQL should be analogous to the way for RDF triples. Table 10.2 contains the textual and visual representations of a SPARQL query. Again, the visual representation is a snapshot of our SPARQL visual editor. Using check and combo boxes in the visualization, users can immediately see the features of SPARQL without the need of learning the complex SPARQL syntax.

Our SPARQL visual editor allows the modification of queries in a similar way for RDF triples: using $+$ to add a triple pattern or a new modifier; using $-$ for removing. Furthermore, users' modification is parsed and checked immediately after each input, and error information is prompted to users. Consequently, our editor ensures users to construct syntactically correct queries.

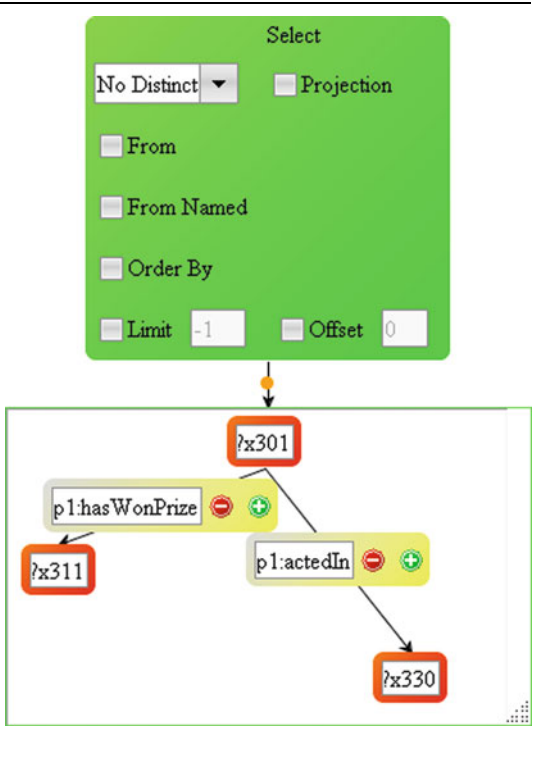
10.5 Browser-Like Query Creation

The browser-like query creation (see Fig. 10.1) starts with a query for all data or with a query transformed from a visual or textual query. The result of the current

Table 10.1 Example RDF Data `Records.rdf`

Textual representation	Visual representation
<code><a> p1:hasWonPrize <p>.</code>	
<code><a> p1:actedIn <f>.</code>	

Table 10.2 Example SPARQL Query `DLCRecords.sparql`

Textual representation	Visual representation
<pre> PREFIX p1: <http://www.pl/> SELECT * WHERE { ?x301 p1:hasWonPrize ?x311. ?x301 p1:actedIn ?x330. } </pre>	 <p>The visual representation consists of two parts. The top part is a green control panel titled "Select" with the following options: "No Distinct" (dropdown), "Projection" (checkbox), "From" (checkbox), "From Named" (checkbox), "Order By" (checkbox), "Limit" (input field with value -1), and "Offset" (input field with value 0). The bottom part is a graph showing a central node "?x301" (in a red box) connected to two nodes: "p1:hasWonPrize" (in a yellow box) and "p1:actedIn" (in a yellow box). The "p1:hasWonPrize" node is connected to a node "?x311" (in a red box). The "p1:actedIn" node is connected to a node "?x330" (in a red box). Each node has a red minus sign and a green plus sign button next to it.</p>

query is presented in the form of a result table. The result table contains buttons to further modify the query. For example, the “Rename” button allows renaming the column name of the result table, that is, renaming the corresponding variable name in the query. The button “Sort” supports to sort the result according to a certain column, the button “Exclude” to hide the column and the button “Refine” to refine the query based on the values of this column. We will explain the refinement of queries in later sections. The “Filter” button supports to filter the result table according to the values of a column and excludes the other rows in the result table. The browser-like query creation also allows to go back to previously created queries (button “<”) and then to return to the later created queries (button “>”).

The advantage of the browser-like query creation is that it is very easy to use, features are always visible, and no knowledge of the SPARQL syntax is necessary. Furthermore, users often like to work on the concrete data rather than on an abstract query seeing immediately the effect of the query modification regarding a correct query result as hint for a correct query. The disadvantage is that not all features of SPARQL can be provided by an easy graphical user interface like it is the case

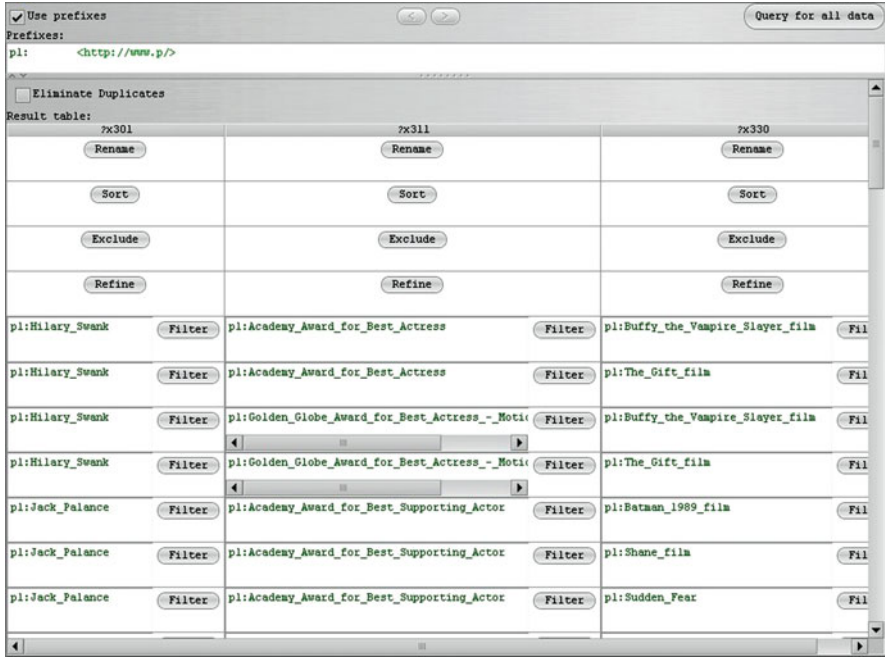


Fig. 10.1 The browser-like query creation

during visual query editing. For example, the union of two (sub) queries cannot be created with the browser-like query creation approach. However, such a query can be taken over from the visual query editor and the query can be further modified with the browser-like query creation approach.

10.6 Generating Condensed Data View

A condensed data view provides an efficient way to see the data structure and to get an overview of the original large datasets. Using the condensed view, one can easily determine the exact queries for the concrete purposes.

A condensed view is generated by integrating the nodes of the RDF graph with certain common features into one node. We use a stepwise approach to condensing data. The main steps are as follows:

Step 1. integrating all *leafs* with the same subject into one node (see Fig. 10.2a). A leaf is an object with only one incoming and without outgoing edges. We condense first all leafs in order to condense the information of a subject into one node, which is otherwise only displayed in a space-consuming way, and to preserve the information of long paths in the data.

Step 2. integrating all objects with the same subject and predicate into one node (see Fig. 10.2b). The structure of the data related to objects with the same predicate

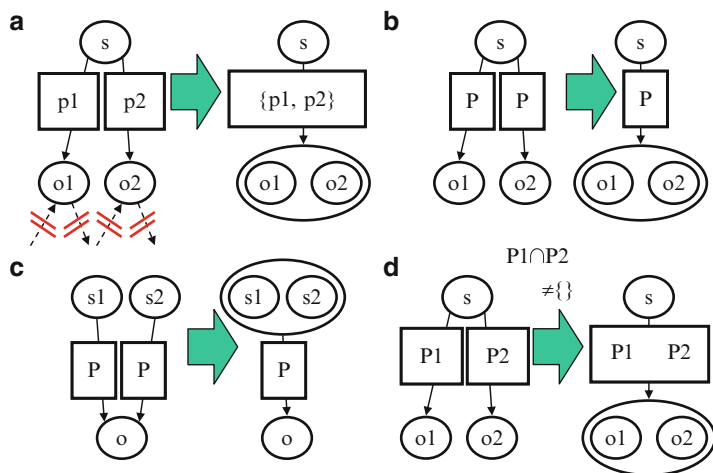


Fig. 10.2 Condensing steps

are often the same or similar, such that this condensing strategy often yields good results.

Step 3. integrating all subjects with the same predicates and objects into one node (see Fig. 10.2c). Similar remarks as for Step 2 also apply to the quality of condensing for this step.

Step 4. integrating the objects with the same subject and at least one predicate in common into one node (see Fig. 10.2d). This step condenses further allowing a sparse representation of the data.

Figure 10.2 describes the condensing steps and Fig. 10.3 presents the condensed view of the Yago data (Suchanek et al. 2007). All the four steps can be processed in reasonable time even for large datasets.

10.7 Refining Queries

Our VQS efficiently supports refinement of queries by a *suggestion functionality* in our SPARQL visual editor. The refined queries can retrieve more exact results for efficient data analysis.

In order to refine a query, the user selects the related node from the SPARQL graph in the SPARQL visual editor, for example, the node for the variable $?x330$ in Table 10.2, or presses the button “Refine” in the browser-like query creation frame (see Fig. 10.1). Afterward, we create two suggestion-computing queries (see Sect. 10.8) and evaluate them on the RDF data. According to the evaluation results of these queries, our system can suggest the triple patterns related with the node for $?x330$. Among these suggested triple patterns, those with $?x330$ as object are called *preceding triple patterns*, and those with $?x330$ as subject are called *succeeding*

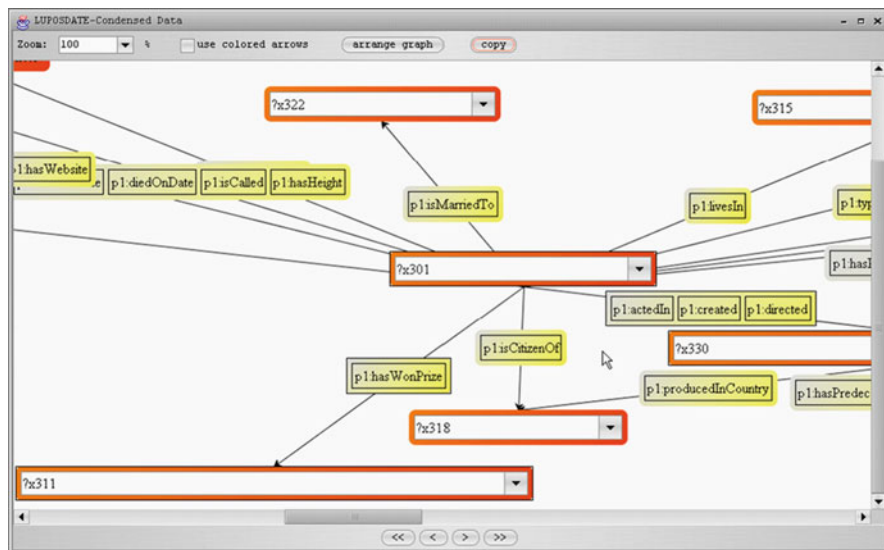


Fig. 10.3 Condensed data view of Yago data

triple patterns. We will explain how to retrieve the suggested triple patterns in detail in Sect. 10.9.

Our system will automatically insert the suggested triple patterns selected by users into the query. Figure 10.4 presents the suggestion dialog for the node ?x330.

10.8 Query Formulation Demo

Assume that there is a kind of users, who is neither familiar with the structure of the data nor with the SPARQL query language for whatever reasons. We want to demonstrate by a simple example how such kind of users can easily create the queries by using our VQS.

In order to formulate queries for, for example, Yago data, a user first browses the condensed view of the Yago data. In order to obtain the condensed view, the user selects the Yago data and then clicks “Condense data” from the main window. A condensed view window is popped, which contains the condensed data views from different condensing steps. Figure 10.3 presents a condensed data view of the Yago data (Suchanek et al. 2007).

The user is interested in the actors, which have won a prize, and in the films, in which the actors acted. Thus, the user selects the corresponding nodes in the condensed view and copies them by clicking *Copy* in the condensed view window. Afterwards, the user opens the query editor from the main window, chooses a

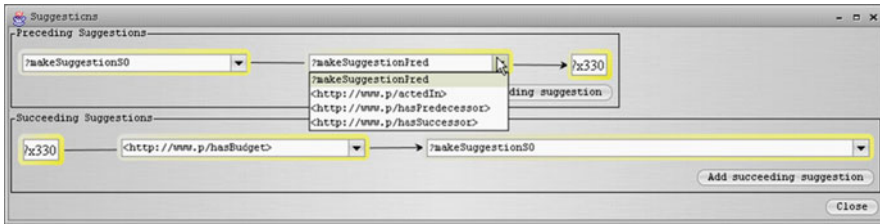


Fig. 10.4 Suggestion dialog for the variable `?x330`

SELECT query, and pastes the nodes into the query editor by clicking *Edit – Paste*. The generated visual query is described in Table 10.2 and its query result is displayed in the browser-like query creation frame (see Fig. 10.1).

After investigating the query result, the user wants to have more information about the films, and thus the user should refine the query. In order to refine queries easily, the user uses our suggestion functionality in either the browser-like query creation approach (button “Refine” in the column for variable `?x330`) or the SPARQL visual editor. In the SPARQL visual editor, the user needs to call *Edit – Make suggestions*. After that, the user clicks the node for the variable `?x330` (which selects films), and a suggestion window for query refinement is popped (see Fig. 10.4).

From the suggestion dialog, the user chooses a succeeding triple pattern, `?x330 p1:hasBudget ?b`. The triple pattern is automatically integrated into the edited query in the visual editor as well as in the browser-like query creation frame after clicking the corresponding button *Add succeeding suggestion*.

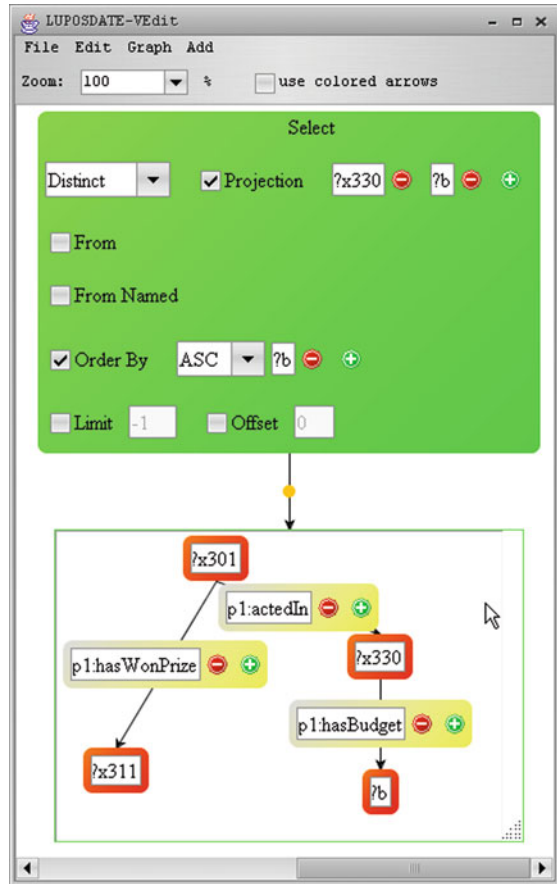
Furthermore, the user also wants to see the querying result sorted according the budgets. By only looking at the visual editor, the user can easily see that the feature *Order By* is the one wanted. In the browser-like query creation frame it is even more obvious to click on the “Sort” button in the column for the variable `?b`. Figure 10.5 presents the final visual query.

10.9 Computation of Suggested Triple Patterns for Query Refinement

Our VQS supports the suggesting functionality for the refinement and extension of queries. Our suggesting functionality computes a group of triple patterns related to the node marked by the user. These triple patterns are recommended to the user for query extension and refinement. In order to determine this group of triple patterns, we first create two SPARQL queries, which we name *suggestion-computing queries*.

For example, a user wants the suggested triple patterns related to the node `?x330` in Table 10.2. In order to compute these triple patterns, we first construct the following two suggestion-computing queries Q1 and Q2:

Fig. 10.5 Final visual query



Q1:	Q2:
<pre>PREFIX p1: <http://www.pl/> SELECT DISTINCT ?x330 ?p ?so WHERE { ?x301 p1:hasWonPrize ?x311. ?x301 p1:actedIn ?x330. ?x330 ?p ?so. }</pre>	<pre>PREFIX p1: <http://www.pl/> SELECT DISTINCT ?x330 ?p ?so WHERE { ?x301 p1:hasWonPrize ?x311. ?x301 p1:actedIn ?x330. ?so ?p ?x330. }</pre>

Q1 and Q2 are then evaluated on the given RDF data. From the result of Q1, we can generate the succeeding triple patterns related to the node ?x330. For example, ?x330=<TheFilmII>, ?p=<http://www.pl/hasSuccessor>, and ?so=<TheFilmIII> is one result of Q1; on the basis of this result, we can recommend the following succeeding triple patterns:

- `?x330 <http://www.p/hasSuccessor> <TheFilmIII>`,
- `?x330 ?p <TheFilmIII>`, and
- `?x330 <http://www.p/hasSuccessor> ?so`.

Likewise, from the result of Q2, we can obtain the related preceding triple patterns. Figure 10.4 presents the dialog displaying the suggested triple patterns.

10.10 Summary and Conclusions

In this chapter, we present our solution to formulate SPARQL queries for analyzing the semantic data from, for example, the Social Semantic Web. Such data are often large-scale and are collected from a large number of different sources. By supporting a condensed data view and visual query editing as well as browser-like query creation, our VQS allows the users, who are neither familiar with the SPARQL language nor the data structure, to construct queries easily. We define the visual query language, develop the rules of the transformations between textual and visual representations, and support automatic transformations between the textual and visual representations. Furthermore, our system also provides a precise approach to query refinement based on the query result, thus generating more exact queries.

Chapter 11

Embedded Languages

Abstract The state of the art in programming Semantic Web applications is using complex application programming interfaces of Semantic Web frameworks. Extensive tests are necessary for the detection of errors, although many types of errors could be detected already at compile time. In this chapter, we propose an embedding of Semantic Web languages into the java programming language, such that Semantic Web data and queries are easily integrated into the program code, type safety is guaranteed, and already at compile time, syntax errors of Semantic Web data and queries are reported and unsatisfiable queries are detected.

11.1 Motivation

Programming Semantic Web applications requires to first learn a complex application programming interface of a Semantic Web framework like the one from Jena (Wilkinson et al. 2003). Furthermore, syntax errors in data files and in queries are only detected when the corresponding program instructions are executed, and error messages are reported from the Semantic Web framework. Therefore, extensive tests are necessary for developing stable Semantic Web programs, which must consider as much as possible from every branch in a program execution and from every possible input data. With our contribution, we want to address the following types of errors by a tool for embedding Semantic Web languages into programming languages, which use a static program analysis for avoiding or detecting these types of errors and therefore support the development of more stable programs: Queries with semantic errors do not generate error messages, but lead to unexpected behaviour of the Semantic Web application. Note that queries with semantic errors often return the *empty* result set every time, that is, these queries are *unsatisfiable*, which is a hint for a semantic error in the query. Furthermore, query results often contain data with a certain data type, for example, numeric values, which have to be further processed in data-type-dependent operations such as the summation of numeric values. Therefore, a cast is necessary to a specific programming language-dependant type. If an erroneous query contains values of other types than expected, then cast errors might occur at runtime.

So far suitable tools for embedding Semantic Web data and query languages into existing programming languages, which go beyond the simple use of application programming interfaces and take advantages of an additional program analysis at compile time, are missing. All static program analysis for *detecting errors* in the embedded languages can be processed *at compile time* before the application is really executed by starting the precompiler offering the most possible convenience to the programmer: Some tests may be done without using embedded languages, but then have to be done by using additional tools and copying and pasting code fragments, or by executing the program itself. A static program analysis can detect errors, which – without a static program analysis – may only be detected after running a huge amount of test cases, as the static program analysis considers every branch in the application code. Our tool, which we call *Semantic Web Objects* system (SWOBE), embeds

- The Semantic Web data language RDF/XML
- The query language SPARQL
- The update language SPARUL

into the *java* programming language. SWOBE supports the development of *more stable* Semantic Web applications by

- Providing transparent usage of Semantic Web data and query languages without requiring users to have a deep knowledge of application programming interfaces of Semantic Web frameworks
- Checking the syntax of the embedded languages for the detection of syntax errors already at compile time
- A static type check of embedded data constructs for guaranteeing type safety
- A satisfiability test of embedded queries for the detection of semantic errors in the embedded queries already at compile time
- A determination of the types of query results for guaranteeing type safety and thus avoiding cast errors.

A demonstration of the SWOBE precompiler is available online at [Groppe and Neumann \(2008\)](#), the example SWOBE programs of which cover embedding of RDF/XML constructs [see *Assistant.swb*, *Student.swb*, *University.swb* and *Professor.swb* at [Groppe and Neumann \(2008\)](#)], SPARQL queries (see *TestStudent.swb* and *QueryTest.swb*), and SPARUL queries (see *Assistant.swb*, *Student.swb*, *University.swb*, *Benchmark.swb*, *Professor.swb*, and *UpdateTest.swb*).

11.2 Related Work

We divide contributions to embedded languages into two groups: the embedding of relational query languages such as SQL into programming languages, and the embedding of XML data and its query languages into programming languages.

(continued)

Pascal/R is the first approach for a type safe embedding of a relational query language into a programming language (Schmidt 1977). Pascal/R does not use SQL but a proprietary query language.

There are many contributions that deal with the embedding of SQL into the programming languages C (see IBM 2003; Ingres 2006), Java (see ANSI 1998; Sybase 1998a; Erdmann 2002), Cobol (see Gilmore et al. 1994; Ingres 2006), Ada (see Erdmann 2002; Ingres 2006), Fortran (see Ingres 2006), Basic (see Ingres 2006), Pascal (see Ingres 2006), and functional programming languages (see Buneman and Ohori 1996; Wallace and Runciman 1999; Nagy and Stansifer 2006). Especially, Buneman and Ohori (1996), Bussche and Waller (1999), and Nagy and Stansifer (2006) describe a comprehensive type systems for guaranteeing static type safety for the relational algebra.

Kempa and Linnemann (2003) embed XML and the XPath query language into the object-oriented language Java and call the resultant language XOBE (*XML Objects*). The generated XML data of an XOBE-program are statically checked for type safety with respect to a given schema and the result types of embedded XPath queries are inferred. XOBE_{DBPL} (Schuhart and Linnemann 2005) extends XOBE with a transparent, type-independent, and distributed persistence-mechanism and a statically checked embedded update query language.

XDuce (Hosoya et al. 2005), HaXML (Wallace and Runciman 1999), and XMLambda (Shields and Meijer 2001) deal with the embedding of XML data into functional programming languages with special regard to parametric polymorphism in type systems for XML. <bigwig> (Brabrand et al. 2002) and JWIG (Christensen et al. 2003) allow the embedding of static validated XHTML 1.0, the XML variant of HTML, into C and Java. Furthermore, Christensen et al. (2003) introduce high-level constructs for Web Service programming with an explicit session model.

Serfiotis et al. (2005) describe algorithms for a containment tester and the minimization of RDF/S query patterns. Serfiotis et al. (2005) consider a hierarchy of concepts and properties.

This chapter contains the contributions for embedding Semantic Web data and query languages into existing programming languages of Groppe et al. (2009e).

11.3 Embedding Semantic Web Languages Into JAVA

We first provide an overview of SWOBE and demonstrate the features of SWOBE by an example. Afterward, we explain the ideas and concepts of SWOBE in detail in the following subsections. We refer the interested reader to the specifications of RDF/XML (Beckett 2004), SPARQL (Prud'hommeaux and Seaborne 2008), and SPARUL (Seaborne and Manjunath 2008) for an introduction to the embedded data, query, and update languages.

```

(1) class TestStudent {
(2)   prefix rdf = http://www.w3.org/1999/02/22-rdf-syntax-ns#;
(3)   prefix ub = http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#;
(4)   prefix xsd = http://www.w3.org/2001/XMLSchema#;
(5)   prefix uni = http://www.University.edu/;
(6)   type Course = (URI, rdf:type,ub:Course) ∪ (URI,ub:name,string) ∪
(7)                 (URI,ub:shortName,string)? ∪ (URI, ub:date,date)+;
(8)   type Student = (URI,rdf:type,ub:GraduateStudent) ∪ (URI, ub:name,string) ∪
(9)                 (URI,ub:takesCourse,URI) ∪ Course+ ∪
(10)                ((URI,ub:emailAddress, anyURI) | (URI, ub:telephone, integer))*;
(11)  type Students = Student+;
(12)  public int[] getTelephoneNumbers(){
(13)    String courseURI = "uni:Programming";
(14)    int tel = 0565664732; int tel2 = 0565457893;
(15)    String email = "Henry@hotmail.com";
(16)    rdف<Course> course = <ub:Course rdf:about=#courseURI#>
(17)                <ub:name rdf:datatype="xsd:string">Programming</ub:name>
(18)                <ub:shortName rdf:datatype="xsd:string">Prog</ub:shortName>
(19)                <ub:date rdf:datatype="xsd:date">2008-02-09</ub:date>
(20)                <ub:date rdf:datatype="xsd:date">2008-02-16</ub:date>
(21)                </ub:Course>;
(22)    rdف<Students> students = <rdf:RDF><ub:GraduateStudent rdf:about="uni:MatrNr552662">
(23)                <ub:name rdf:datatype="xsd:string">Henry Schmidt</ub:name>
(24)                <ub:takesCourse>#course#</ub:takesCourse>
(25)                <ub:telephone rdf:datatype="xsd:integer">#tel#</ub:telephone>
(26)                <ub:emailAddress rdf:datatype="xsd:anyURI">#email#</ub:emailAddress>
(27)                </ub:GraduateStudent>
(28)                <ub:GraduateStudent rdf:about="uni:MatrNr552663">
(29)                <ub:name rdf:datatype="xsd:string">Anne Mustermann</ub:name>
(30)                <ub:takesCourse>#course#</ub:takesCourse>
(31)                <ub:telephone rdf:datatype="xsd:integer">#tel2#</ub:telephone>
(32)                </ub:GraduateStudent> </rdf:RDF>;
(33)    SparqlIterator query = sparql ( SELECT ?Y ?Z FROM #students#
(34)                                   WHERE {
(35)                                     ?X rdf:type ub:GraduateStudent .
(36)                                     ?X ub:telephone ?Y .
(37)                                     ?X ub:takesCourse ?V .
(38)                                     { ?V ub:shortName ?Z . }
(39)                                   UNION
(40)                                     { ?V ub:name ?Z . } } );
(41)    int[] telnumber = new int[query.getRowNumber()];
(42)    String[] name = new String[query.getRowNumber()];
(43)    int i = 0;
(44)    while(query.hasNext()){
(45)      Result res = query.next();
(46)      telnumber[i] = res.getY();
(47)      if(res.getZ() != null) name[i] = res.getZ();
(48)      i++;
(49)    query.close();
(50)    return telnumber; } }

```

Fig. 11.1 A SWOBE example program. The bold facepart contains specific SWOBE expressions

Figure 11.1 contains an example SWOBE program, which uses the RDF format to describe information about students and the courses they take [see lines (16–32) of Fig. 11.1]. The type of the embedded RDF data is defined in lines (6–11) of Fig. 11.1. An embedded SPARQL query [see lines (33–39) of Fig. 11.1] asks for the telephone number of those students, which take at least one course. Additionally, the name and the short name of the courses taken by the student are contained in the query result. Afterward, the telephone numbers of the students and the names or short names respectively of the courses are stored in arrays by iterating through the query result [see lines (41–48) of Fig. 11.1].

Figure 11.2 depicts the architecture of the SWOBE precompiler.

The SWOBE precompiler first parses the SWOBE program according to the Java 1.6 grammar with the extension of embedded RDF/XML constructs (e.g., lines (16–21) and lines (22–32) of Fig. 11.1), prefix declarations (e.g., lines (2–5) of Fig. 11.1), and SPARQL/-UL queries (e.g., the SPARQL query in lines (33–39) of Fig. 11.1). Syntax errors of embedded RDF/XML constructs and SPARQL/-UL queries are already detected and reported in this phase at compile time, which are otherwise – in the case of using Java 1.6 with Semantic Web application programming interfaces – only detected at runtime maybe after running extensive tests.

We assume S to be the type of the right side of an assignment of RDF data to a variable (lines (16–32) of Fig. 11.1) and T to be the variable type. The type system of the SWOBE precompiler then checks whether or not S conforms to T , that is, if S is a subtype of T .

The satisfiability tester of the SWOBE precompiler afterward checks whether the result of the embedded SPARQL/-UL queries (e.g., lines (33–39) of Fig. 11.1 for a SPARQL query) is empty for any input based on the type of the input data.

The SWOBE precompiler then determines the java types of the results of the embedded SPARQL queries. The SWOBE precompiler uses the java types for

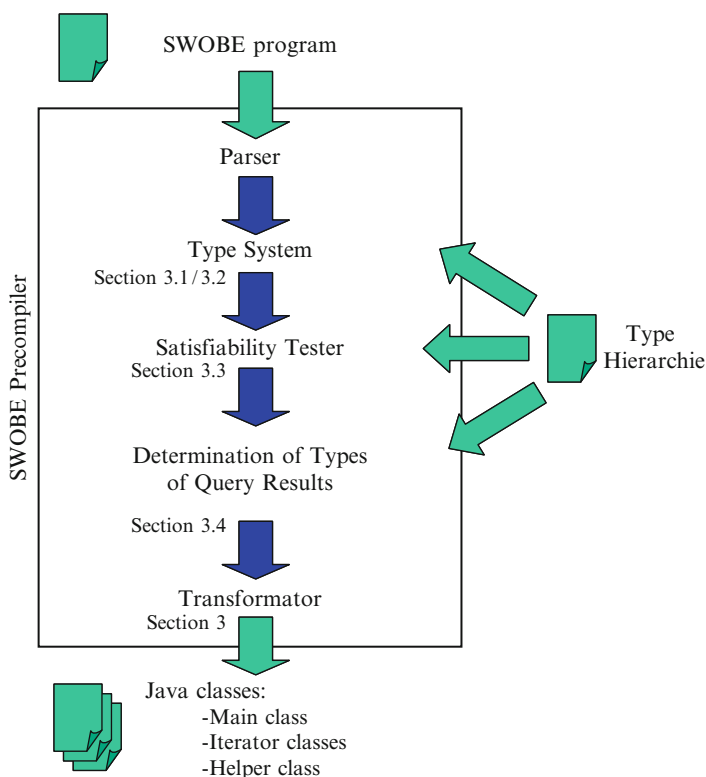


Fig. 11.2 The architecture of the SWOBE precompiler

generating the special iterators for query results. In the example of Figure 11.1, the SWOBE precompiler generates an iterator for the result of the SPARQL query in lines (33–39). The iterator contains the special methods `int getY()` and `String getZ()` for accessing the results of the variables `Y` and `Z`, respectively. Note that the SWOBE precompiler also determines the result type of `Y` to be `int` and of `Z` to be `String` based on the type `Students` of the input data `#students#` of the SPARQL query.

At last the SWOBE precompiler transforms the SWOBE program into java classes. The generated java classes use the application programming interfaces (API) of existing Semantic Web frameworks. Our SWOBE precompiler currently supports the API of the widely used Jena Semantic Web framework (Wilkinson et al. 2003), but can be easily tailored to support APIs of other Semantic Web frameworks. The transformed java classes are the main class corresponding to the SWOBE program, some iterator classes for query results like query in Fig. 11.1, and helper classes, methods of which are called by the main class.

11.3.1 The Type System

RDFS/OWL ontologies are designed to handle incomplete information. This design decision leads to the following two phenomena:

1. OWL and RDFS ontologies do not provide any constraints for entities that are not typed. For example, the triple $(s, \text{rdf:type}, c)$ types the s entity to be of class c . If an ontology is given, which has constraints for members of the class c like a maximal cardinality one of a property `color` for entities of the class c , then the triples $(s, \text{color}, \text{blue})$ and $(s, \text{color}, \text{red})$ are inconsistent with this ontology. However, no entity, not `_:b1` and not `_:b2`, is typed in the triple set $\{(_:b1, \text{uni:name}, \text{“UniversityOfLübeck”}), (_:b1, \text{uni:institute}, _:b2), (_:b2, \text{uni:name}, \text{“IFIS”})\}$, such that no any ontology can impose constraints on the triples of this triple set. Thus, this triple set conforms to any ontology.
2. Even if an entity is typed, a given ontology does not impose any constraints for properties and objects, which are not listed in the ontology. Thus, a fact (s, p, o) is still consistent with a given RDFS/OWL ontology, even when $(s, \text{rdf:type}, c)$ is true and there are no constraints given in the RDFS/OWL ontology about the object o or predicate p for members of the class c .

However, if the types S and T are described by ontologies, the check whether or not a type S is a subtype of another type T would not consider the triples not described by an ontology according to phenomena (1) and (2). In this case, we could only state that S is a subtype of T *except* of triples according to phenomena (1) and (2). Additionally, the satisfiability test of embedded SPARQL queries based on a given type for the input triples would detect only *maybe* unsatisfiable queries, which are unsatisfiable for triples without those of phenomena (1) and (2). However, we cannot guarantee the exclusion of triples according to phenomena (1) and

```

TypeDefinition ::= OrType | "ANY"
OrType ::= UnionType ( "|" UnionType )*
UnionType ::= TripleExpr ("|" "*" "?" )?
              ("|" "*" "?" )?
TripleExpr ::= "(" ElemExpr "," ElemExpr "," ElemExpr ")"
              | "<IDENTIFIER> | "(" OrType ")"
ElemExpr ::= Value ( "(" Value "(" <STRING_LITERAL>
                  ("|" <STRING_LITERAL>* ")" ) ) )?

```

where <IDENTIFIER> represents an identifier (the name of a named type), Value a basic type and <STRING_LITERAL> a string literal.

Fig. 11.3 EBNF rules for defining types of embedded RDF data

(2). Furthermore, the determination of the query result types based on a given type for the input triples fails to consider the triples according to phenomena (1) and (2).

Therefore, we propose to use a type system, which avoids the two aforementioned phenomena of RDFS/OWL ontologies. Note that our type system supports incomplete information by allowing any triples if explicitly stated.

Our developed language for defining the types of embedded RDF data conforms to the EBNF rules of Fig. 11.3.

We can define the types of triple sets with this language. If the type is ANY, then there are no restrictions to the triple set. Types for single triples consist of three basic types for the subject, predicate, and object of a triple. A basic type is a concrete URI, literal, or an XML Schema data type. We can exclude values for basic types; for example, string \ “Fritz” allows all strings except “Fritz”. Furthermore, if A and B are types for triple sets, then $A \mid B$, $A \cup B$, A^* , A^+ , $A^?$ and (A) are also types for triple sets: $A \mid B$ allows triples of type A or B. A triple set V conforms to a type $A \cup B$ if triple sets V_1 and V_2 exist, such that $V_1 \cap V_2 = \{ \}$ and $V = V_1 \cup V_2$ and V_1 conforms to type A and V_2 conforms to type B. Arbitrary repetitions can be expressed by using A^* for including zero repetitions, A^+ for at least one repetition and $A^?$ for zero or one repetition. Bracketed expressions (A) allow specifying explicit priorities between the operators of a type, for example, $(A \cup B)^*$. References to named types may be used in a type for reusing already defined types.

We further allow types for predicate-object-lists, triples of which have the same subject, and for object-lists, triples of which have the same subject and the same predicate. We do not present these extensions here due to the simplicity of presentation, as these extensions complicate the algorithms for subtype tests due to more cases to be considered, but do not show new insights for subtype tests.

In the example of Fig. 11.1, the type definition Course [see lines (6–7) of Fig. 11.1] describes the RDF data about a course at a university. The type definition Student [see lines (8–10) of Fig. 11.1] describes the RDF data about a student in a university, and the type definition Students [see line (11) of Fig. 11.1] describes a group of students.

11.3.2 Subtype Test

Whenever a variable is assigned with RDF data [e.g., lines (22–32) of Fig. 11.1], we can determine the type of this assigned RDF data. The type S of assigned RDF data [e.g., the assigned RDF data in lines (22–32) of Fig. 11.1] is the union of the types of the triples, which are generated, and the types of embedded variables [e.g., `#course` in line (24)] containing RDF data. The subtype test is used for checking whether or not the type S of assigned RDF data [e.g., the assigned data in lines (22–32) of Fig. 11.1] conforms to an expected type T [e.g., the type `rdf<Students>` of the variable `students` in line (11) of Fig. 11.1] for the content of the assigned variable. In general, the subtype test checks whether or not a type S is a subtype of another type T ; that is, the subtype test checks whether or not all possible input data, which are of type S , are also of type T .

We first simplify the type definition T and S according to the formulas presented in Fig. 11.4, such that superfluous brackets are eliminated and subexpressions of the form $A \theta_1 \theta_2$, where $\theta_1, \theta_2 \in \{+, *, ?\}$, are transformed into subexpressions $A \theta_3$ with one frequency operator θ_3 .

The algorithm `checkSubType(S,T)` (see Fig. 11.5) performs the task of checking if S is a subtype of T . In the special case that the type S describes an empty triple set [see line (2) of Fig. 11.5], it is tested whether or not T allows the empty triple set by using the function `isNullable` (see Fig. 11.6). If T allows any input, then any type is a subtype of T [see line (3) of Fig. 11.5]. If T does not allow any input, but S does,

Fig. 11.4 Simplifying type definitions, where A is a type definition

$$\begin{aligned} A \theta_0 \theta_1 &= A^* \text{ if } \exists i \in \{0, 1\} \wedge \theta_i \in \{+, *\} \wedge \theta_{(i+1)\%2} \in \{?, *\} \\ (A \theta_0) \theta_1 &= A \theta_0 \theta_1, \text{ where } \theta_0, \theta_1 \in \{+, *, ?\} \\ ((A)) &= (A) \\ A \theta \theta &= A \theta, \text{ where } \theta \in \{+, *, ?\} \end{aligned}$$

```
(1) boolean checkSubType(S, T) {
(2)   if(S = ∅) return isNullable(T);
(3)   else if(T = ANY) return true;
(4)   else if(S = ANY) return false;
(5)   else return (∃m = {(s1, t1), ..., (sn, tn)}, where si ∈ SExpr(S) ∧
      ti ∈ SExpr(T): homomorphism(S,T,m)); }
```

Fig. 11.5 Main Algorithm for the test if S is a subtype of T

$$\begin{aligned} \text{isNullable}(A|B) &= \text{isNullable}(A) \vee \text{isNullable}(B) & \text{isNullable}(A *) &= \text{true} \\ \text{isNullable}(A \cup B) &= \text{isNullable}(A) \wedge \text{isNullable}(B) & \text{isNullable}(A ?) &= \text{true} \\ \text{isNullable}(A +) &= \text{isNullable}(A) & \text{isNullable}((s, p, o)) &= \text{false} \\ \text{isNullable}((A)) &= \text{isNullable}(A) & \text{isNullable}(\emptyset) &= \text{true} \end{aligned}$$

Fig. 11.6 Algorithm `isNullable`, where A and B are type definitions, s , p , and o the types of the subject, predicate, and object of a triple

then S cannot be a subtype of T [see line (4) of Fig. 11.5]. Otherwise, we first transform the types S and T into tree representations $tree(S)$ and $tree(T)$ by using a function $tree$. See Fig. 11.7 for the recursive definition of the function $tree$ and an example of its result in Fig. 11.8.

Checking if S is a subtype of T can be reformulated into the problem of finding a homomorphism [see line (5) of Fig. 11.5] from the tree representation $tree(S)$ of S to the tree representation $tree(T)$ of T . We first describe an optimized algorithm for quickly finding such a homomorphism and afterward describe the constraints of the homomorphism in detail for the subtype relation between S and T . We use the

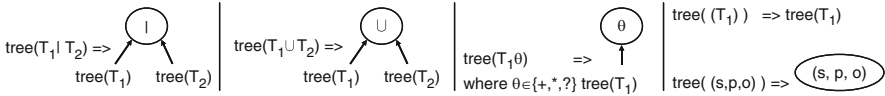


Fig. 11.7 Transforming a type definition into a tree representation. T_1 and T_2 represent type definitions and (s, p, o) the type of a triple

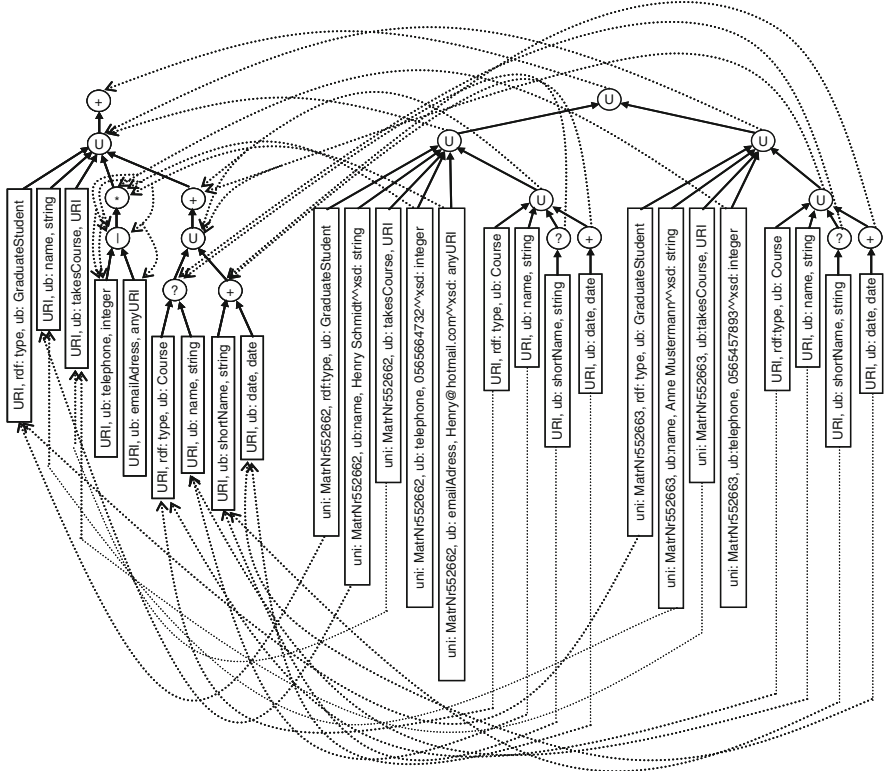


Fig. 11.8 Homomorphism from a type S representing the type of the assigned data in lines (22–32) of Fig. 11.1 on the right side of this figure to a type T representing the type $rdf\langle Students \rangle$ of the variable $students$ in line (11) of Fig. 11.1 on the left side of this figure

tree representation of T and S and a corresponding homomorphism in the example of Fig. 11.8.

A subtype relation is already proved after only one homomorphism relation between two types is found.

We propose to search for a homomorphism between two types in two phases. The first phase determines all single candidate mappings from subexpressions of S to subexpressions of T. The candidate mappings can be determined very fast in the time $O(|S|*|T|)$, where $|V|$ represents the length of a type V, that is, the number of subexpressions of V. The efficient algorithm visits the tree representation of S bottom-up in order to find suitable subexpressions in T. During visiting S bottom-up, the algorithm considers already found mappings for the current node's children in the tree representation of S to nodes in the tree representation of T.

Afterward, we determine suitable subsets of the candidate mappings, which will be checked by the algorithm homomorphism (see Fig. 11.9) and this is explained in the next paragraph. This second phase is designed to quickly exclude unsuitable subsets of the candidate mappings. Furthermore, we first check those subsets of candidate mappings, which are promising to be a homomorphism. As a subexpression in S must not be mapped to two different subexpressions in T, we exclude this kind of subsets of the candidate mappings. In order to further exclude subsets of the candidate mappings earlier, we consider afterward candidate mappings for a subexpression in S, which is mapped to a minimum number of subexpressions in T in the remaining set of candidate mappings. In order to abort the search for a homomorphism in unsuitable subsets of the candidate mappings as early as possible, we do *not* consider a possible subset of candidate mappings further if a mapping (s, t) is in the subset C of candidate mappings, where one child s_1 of s and a mapping $(s_1, t_1) \in C$ exists such that $t \neq t_1$ and t_1 is not a subexpression of t. In this case, the constraints imposed by the homomorphism of the subtype relation cannot be fulfilled anymore.

The function homomorphism (see Fig. 11.9) expects the types T and S and a mapping m as input. The function first checks if a subexpression of S is mapped to

```
(1) boolean homomorphism(S, T, m) {
(2)   if (  $\exists s \in \text{SEExpr}(S): (s, t_1) \in m \wedge (s, t_2) \in m \wedge t_1 \neq t_2 \wedge$ 
(3)      $!(\exists t_1', \dots, t_n' \in \text{SEExpr}(T) : ((t_1' = t_1 \wedge t_n' = t_2) \vee (t_1' = t_2 \wedge t_n' = t_1)) \wedge$ 
(4)      $\forall i \in \{1, \dots, n-1\}: ((t_i' = t_{i+1}' \theta, \text{ where } \theta \in \{+, *, ?\}) \vee (t_i' = t_{i+1}' | t' \vee t_i' = t' | t_{i+1}',$ 
(5)      $\text{ where } t' \in \text{SEExpr}(T)) \vee (t_i' = t_{i+1}' \cup t' \vee t_i' = t' \cup t_{i+1}',$ 
(6)      $\text{ where isNullable}(t') \wedge t' \in \text{SEExpr}(T))))$  return false;
(7)   if (  $(S, T) \in m \wedge \forall s \in \text{SEExpr}(S): \exists (s, t) \in m \wedge$ 
(8)      $\forall (s, t) \in m: \text{isMappingOfHomomorphism}(s, t, T, m)$  ) return true;
(9)   else return false; }
```

Fig. 11.9 Function homomorphism for checking if m describes a homomorphism from S to T

only one subexpression of T [see line (2) of Fig. 11.9], as otherwise the mapping would be ambiguous with some exceptions: One exception is a subexpression with a frequency operator [see line (4) of Fig. 11.9]: if s is a subtype of t , then s is also a subtype of $t \theta$, where $\theta \in \{+, *, ?\}$. Another exception is a subexpression with an or-operator: if s is a subtype of t , then s is also a subtype of $t \mid t'$ [see line (4) of Fig. 11.9]. The last exception is a subexpression with a union-operator with at least one operand, which allows the empty expression [see line (5) of Fig. 11.9]: if s is a subtype of t , then s can be also a subtype of $t \cup t'$ with $\text{isNullable}(t)$ or $\text{isNullable}(t')$ holds and we later check whether or not s is really a subtype of $t \cup t'$. Afterward, the function homomorphism checks if the type S is mapped to the type T and if all subexpressions $\text{SEExpr}(S)$ (see Fig. 11.10) of S are mapped to subexpressions of T , fulfilling further constraints checked in the function $\text{isMappingOfHomomorphism}$ for each single mapping entry [see lines (7–8) of Fig. 11.9].

The function $\text{isMappingOfHomomorphism}$ checks if the given mapping m from type S to type T describes a part of a homomorphism from S to T . If S is composed of two subtypes S_1 and S_2 in an or-relation $S_1 \mid S_2$, then there should exist mappings from S_1 and S_2 to T [see line (2) of Fig. 11.11] for a subtype relation. If T is

$$\begin{aligned} \text{SEExpr}(A_1 \mid \dots \mid A_n) &= \{(A_1 \mid \dots \mid A_n)\} \cup \text{SEExpr}(A_1) \cup \dots \cup \text{SEExpr}(A_n), \text{ where } n \geq 2 \\ \text{SEExpr}(A_1 \cup \dots \cup A_n) &= \{(A_1 \cup \dots \cup A_n)\} \cup \text{SEExpr}(A_1) \cup \dots \cup \text{SEExpr}(A_n), \text{ where } n \geq 2 \\ \text{SEExpr}(A \theta) &= \{(A \theta)\} \cup \text{SEExpr}(A), \text{ where } \theta \in \{+, *, ?\} \\ \text{SEExpr}((s, p, o)) &= \{(s, p, o)\} \\ \text{SEExpr}(A) &= \text{SEExpr}(A) \end{aligned}$$

Fig. 11.10 Function SEExpr , where A, A_1, \dots, A_n are type definitions and (s, p, o) is the type of a triple

- (1) boolean $\text{isMappingOfHomomorphism}(s, t, T, m)$ {
- (2) if $(s = s_1 \mid s_2)$ return $((s_1, t) \in m) \wedge ((s_2, t) \in m)$;
- (3) else if $(t = t_1 \mid t_2)$ return $((s, t_1) \in m) \vee ((s, t_2) \in m)$;
- (4) else if $(s = s_1 \cup s_2 \cup \dots \cup s_n \wedge t = t_1 \cup t_2 \cup \dots \cup t_p)$ {
- (5) return $(\exists S'_1, \dots, S'_k: ((\text{repetition}(t, T) \wedge \exists q \in \mathbb{N}: k = p^*q) \vee (k = p)) \wedge$
- (6) $\forall i \in \{1, \dots, k\}: S'_i \subseteq \{s_1, \dots, s_n\} \wedge !(\exists j \in \{1, \dots, k\} - \{i\}: S'_i \cap S'_j \neq \emptyset) \wedge$
- (7) $((\cup_{S'_i \in S'_1} S'_i, t_{((i-1) \bmod p)+1}) \in m \vee (\text{isNullable}(t_{((i-1) \bmod p)+1}) \wedge S'_i = () \wedge$
- (8) $(\text{isNullable}(t_{((i-1) \bmod p)+1}) \vee S'_i \neq \emptyset))$);
- (9) } else if $(s = (s_1, p_1, o_1) \wedge t = (s_2, p_2, o_2))$ {
- (10) return $\text{isSubtype}(s_2, s_1) \wedge \text{isSubtype}(p_2, p_1) \wedge \text{isSubtype}(o_2, o_1)$;
- (11) } else if $(t = t_1 \theta_1, \text{ where } \theta_1 \in \{+, *, ?\})$ {
- (12) if $(\text{freq}(s) \neq \text{freq}(t) \wedge !(\text{freq}(s) <_t \text{freq}(t)))$ return false;
- (13) if $(s = s_1 \theta_2, \text{ where } \theta_2 \in \{+, *, ?\})$ return $((s_1, t_1) \in m)$;
- (14) else return $((s, t_1) \in m)$; }

Fig. 11.11 Function $\text{isMappingOfHomomorphism}$ for checking if m describes a homomorphism from s to t

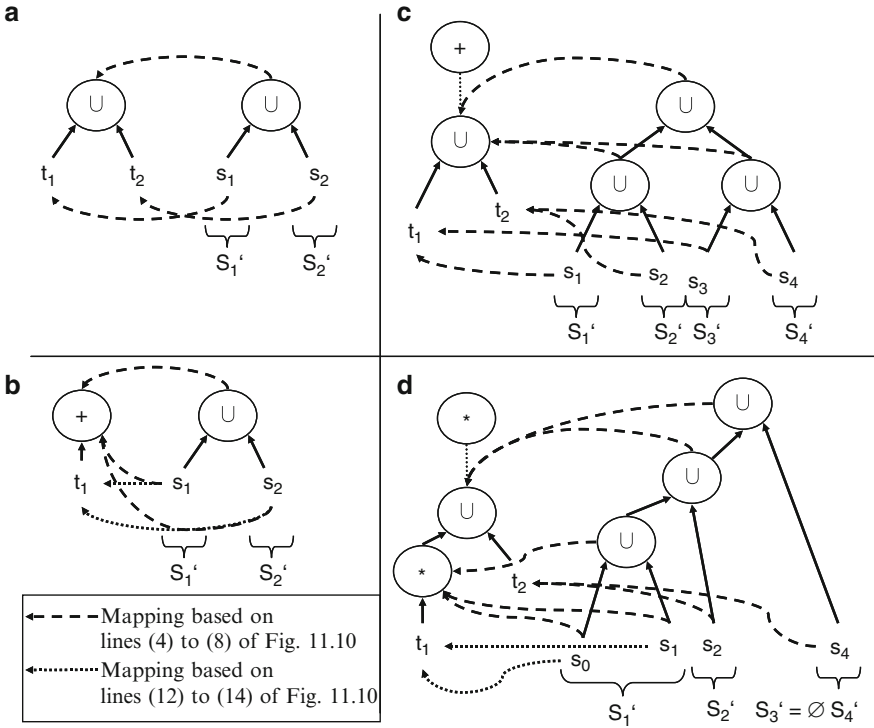


Fig. 11.12 Different examples for the subtype test when checking union operations. Here, t_1, \dots, t_p are subexpressions of T , s_1, \dots, s_n are subexpressions of S , S_1', \dots, S_k' are sets of the decomposition, s_i is a subtype of $t_{((i-1) \bmod p)+1}$, and s_0 is a subtype of t_1

composed of two subtypes T_1 and T_2 in an or-relation $T_1 \mid T_2$, then there should exist at least one mapping from S to T_1 or T_2 [see line (3) of Fig. 11.11] for a subtype relation.

We present examples for subtype tests between types containing union operations in Fig. 11.12. Example (a) in Fig. 11.12 is the simplest case. In this example, S and T are the union of two subtypes, and each union-operand of S must be a subtype of another union-operand of T . The pictures of the examples (b)–(d) in Fig. 11.12 become more complicated since the union-operator of T can be arbitrarily repeated (they are in the scope of a $*$ or a $+$ operator). In the example (b), several union-operands of S must be subtypes of the same union-operand of T . If T consists of a union of types, which can be arbitrarily repeated (see e.g., (c)), and the union-operator in S has more operands than the union-operator in T , then several pairwise disjoint decompositions of the union-operands of S must be subtypes of T . Furthermore, if an operand of the union-operator of T can be arbitrarily repeated like in example (d), then several union-operands of S can be the subtype of this arbitrarily repeatable union-operand of T . If an operand of the union-operator of T allows the empty triple set like in example (d), then no union-operand of S may be the

$$\text{repetition}(t, T) = (\exists t' \in \text{SEExpr}(T) : t \in \text{SEExpr}(t') \wedge t' = t' \theta \wedge \theta \in \{ +, * \})$$

Fig. 11.13 Function $\text{repetition}(t, T)$ for the determination whether or not the subexpression t is part of a subexpression of the type T , which can be arbitrarily repeated

Fig. 11.14 Function freq , where A and B are type definitions, s is the type of the subject, p of the predicate, and o of the object of a triple

$$\begin{aligned} \text{freq}(A \theta) &= \theta, \text{ where } \theta \in \{ +, *, ? \} \\ \text{freq}(A \cup B) &= \text{One} \\ \text{freq}(A \mid B) &= \text{One} \\ \text{freq}((s, p, o)) &= \text{One} \end{aligned}$$

subtype of this union-operand of T . In order to deal with all these cases, we check the following condition: If S and T are composed of types in a union-relation [line (4) of Fig. 11.11], then there should exist a pairwise disjoint decomposition of the operands [line (6) of Fig. 11.11] of the union operator, such that the number of decompositions is the same as the number of union-operands of T or is a factor of the number of union-operands of T in the case that the subexpression of the union-operator can be arbitrarily repeated [line (5) of Fig. 11.11] as it is in the scope of a $*$ or a $+$ operator (see Fig. 11.13). Furthermore, all these decompositions must be mapped to the corresponding union-operand of T according to the number of repetitions [line (7) of Fig. 11.11] and all union-operands of T should have a candidate mapping or should allow the empty triple set [line (8); Fig. 11.11].

If the types S and T describe constraints for single triples, then each triple element, that is, the subject, predicate, and object, of S must be a subtype of the corresponding triple element of T [lines (9) and (10) of Fig. 11.11]; for example, `xsd:long` is a subtype of `xsd:decimal` according to the type hierarchy of XML Schema data types (see Peterson et al. (2009)). In the case that T contains an operator $+$, $*$ or $?$ [lines (11–14) of Fig. 11.11], we exclude those candidate mappings, the frequency of which is in conflict with a subtype relation in line (12) of Fig. 11.11. A mapping from S to T is not in conflict with a subtype relation, if they have the same frequency (see Fig. 11.14 for the computation of the frequency of a type) or if the frequency of S is lower than the frequency of T , that is, $\text{freq}(S) <_f \text{freq}(T)$, where the transitive relation $<_f$ holds for $\text{ONE} <_f ? <_f + <_f *$. Afterward, we check if the corresponding subexpressions are in the mapping m (lines (13 and 14) of Fig. 11.11).

11.3.3 Satisfiability Test of Embedded SPARQL and SPARUL Queries

Erroneous queries often return the empty set for any input, and thus queries are unsatisfiable. Therefore, an unsatisfiable query is a hint for errors in the query. Satisfiability tests of queries can (1) warn the user of the errors in queries, and debug SWOBE programs, thus leading to more stable programs, and (2) precompute

the unsatisfiable queries to the empty result at compile time, thus avoiding runtime processing and speeding up the program execution.

Note that SPARUL queries extend the syntax and semantics of SPARQL queries by update queries, such that the below described approaches apply to SPARQL queries *and* SPARUL queries. For checking the satisfiability of embedded queries, we first transform abbreviations of SPARQL/-UL constructs, that is, predicate-object-lists, object-lists, collections, and the a operator, into their equivalent long forms (see Groppe et al. (2009d)). We replace blank nodes by the variables, which are not used somewhere else in the SPARQL/-UL query according to Gutierrez et al. (2004). After this step, each triple pattern of the SPARQL/-UL query has the form $e_1 e_2 e_3$., where e_i is an IRI, a literal (including string and numeric constants) or a variable.

We determine the type D of the input data of the embedded query by a static program analysis. In the example of Fig. 11.15, the satisfiability test and the determination of the query result types are for the embedded SPARQL query in lines (33–39) of Fig. 11.1. The determined type for the variable ?Y is integer and the determined type for the variable ?Z is string. A triple pattern $e_1 e_2 e_3$ is satisfiable, if the type D of the input data contains types of triples, which intersect with $e_1 e_2 e_3$.. We can determine all possible types of variables in the triple pattern by checking all types of triples in D, which intersect with the triple pattern $e_1 e_2 e_3$. Thus, we can use $\text{types}(e_1 e_2 e_3)$ to determine the types of the variables in triple patterns (see Fig. 11.16). If the set of variable types is empty, then this triple pattern is unsatisfiable.

The satisfiability of queries can be determined by the function types of Fig. 11.16. Note that $\text{sat}(\text{Expr}, \text{types}(A))$ is a satisfiability tester for Boolean expressions Expr under data type constraints $\text{types}(A)$ of the variables in Expr. Such a satisfiability tester $\text{sat}(\text{Expr}, \text{types}(A))$ has a high computational complexity (see Cook (1971)). However, the results without using such a satisfiability tester $\text{sat}(\text{Expr}, \text{types}(A))$ for FILTER expressions are typically quite well, such that the application of such a satisfiability tester can be avoided to speed up computation.

If the result of the function types contains the empty set for the types of at least one variable, then the SPARQL/-UL query is unsatisfiable and we can warn the user.

```

SELECT ?Y ?Z FROM #student#
WHERE {
  ?X rdf:type ub: GraduateStudent. } {{?X, {URI}}}
  ?X ub:telephone?Y . } {{?X, {URI}}, (?Y, {integer})} } {{?X, {URI}}, } {{?X, {URI}}, }
  ?X ub:takesCourse?V . } {{?X, {URI}}, (?V,{URI})} } {{?V, {URI}}, } {{?X, {URI}}, }
  { ?V ub:shortName?Z . } } {{?V, {URI}}, (?Z, {string})} } {{?Y, {integer})} } {{?V, {URI}}, }
  UNION } {{?V, {URI}}, } {{?Y, {integer}}, }
  { ?V ub:name?Z . } } {{?V, {URI}}, (?Z, {string})} } {{?Z, {string})} }
}
    
```

Fig. 11.15 Example of the satisfiability test and the determination of the query result types for the embedded SPARQL query in lines (33–39) of Fig. 11.1

$$\begin{aligned}
&\text{intersect}(t1, t2) = (t1 \text{ is a variable} \vee t1 \text{ is subtype of } t2 \vee t2 \text{ is a subtype of } t1) \\
&\text{types}(e_1 e_2 e_3, (e_1', e_2', e_3')) = \{ (e_i, ST) \mid i \in \{1,2,3\} \wedge e_i \text{ is a variable} \wedge \\
&\quad ST = \{e_i'\} \wedge \text{intersect}(e_1, e_1') \wedge \text{intersect}(e_2, e_2') \wedge \text{intersect}(e_3, e_3') \} \\
&\text{types}(e_1 e_2 e_3) = \{(e_i, ST) \mid i \in \{1,2,3\} \wedge e_i \text{ is a variable} \wedge \\
&\quad ST = \cup_{(e_1', e_2', e_3') \in D} \text{types}(e_1 e_2 e_3, (e_1', e_2', e_3'))\} \\
&\text{types}(A B) = \{(e, ST) \mid (e, ST') \in \text{types}(A) \wedge (e, ST'') \in \text{types}(B) \wedge ST = \{t \mid t \text{ is a super type of} \\
&\quad t' \in ST' \wedge t \text{ is a super type of } t'' \in ST''\}\}, \text{ where } A \text{ and } B \text{ are group graph patterns} \\
&\text{types}(\{A\}) = \text{types}(A), \text{ where } A \text{ is a group graph pattern} \\
&\text{types}(A \text{ OPTIONAL } B) = \text{types}(A) \cup \{ (e, ST) \mid (e, ST') \notin \text{types}(A) \wedge (e, ST) \in \text{types}(B) \} \\
&\text{types}(A \text{ UNION } B) = \{(e, ST) \mid (e, ST') \in \text{types}(A) \wedge (e, ST'') \in \text{types}(B) \wedge ST = ST' \cup ST'' \vee \\
&\quad ((e, ST) \in \text{types}(A) \wedge (e, ST'') \notin \text{types}(B)) \vee ((e, ST') \notin \text{types}(A) \wedge (e, ST) \in \text{types}(B))\} \\
&\text{types}(A \text{ FILTER}(\text{Expr})) = \begin{cases} \text{types}(A) & \text{if sat}(\text{Expr}, \text{types}(A)) \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 11.16 The function types determining the types of variables in a SPARQL query, and the function intersect checking if elements of a triple type and a triple pattern intersect. A and B are SPARQL subexpressions, and e_i is the type of an element of a triple type or of a triple pattern

11.3.4 Determination of the Query Result Types

We have already determined the possible types of the result of an embedded SPARQL query when testing the satisfiability. For returning the result by an iterator, we have to determine a super type of these possible types. This super type is then the return type for the iterator method.

We present in Fig. 11.15 the determination of the query result types for the embedded SPARQL query in lines (33–39) of Fig. 11.1. The type for the variable ?Y in the query result is integer and the type for the variable ?Z is string.

Once we have determined the return type, we can generate code for a query result iterator with this return type, such that the type system of java guarantees type safety for the usages of the result. In the example of Fig. 11.15, the java type for the variable ?Y is int and the java type of the variable ?Z is string.

11.4 Summary and Conclusions

We have proposed an approach to supporting the development of more stable Semantic Web applications by embedding the Semantic Web languages RDF/XML, SPARQL, and SPARUL into the java programming language.

Our Semantic Web *Objects* system (SWOBE) uses a static program analysis in order to guarantee type safety, detect unsatisfiable SPARQL/-UL queries, and determine the types of query results at compile time. In this way, we avoid runtime errors and unexpected behavior of the Semantic Web application. Our implementation of the SWOBE system shows the advantages of our approach as a programming tool.

Chapter 12

Comparison of the XML and Semantic Web Worlds

Abstract XML and the Semantic Web cover many specifications of languages for the web, which can be used for similar applications. We compare both worlds, the Semantic Web one and the XML one, and show how to transform queries and data from one to the other. We also provide a comprehensive performance analysis for translated queries.

12.1 Introduction

XML (W3C 2010b) and the Semantic Web (W3C 2010a) are both initiated and supported by the W3C, where XML has an older history (see Fig. 12.1) than the Semantic Web.

The direct predecessor of XML data (W3C 2004b), *Standard Generalized Markup Language* (SGML) (ISO 1986), has been already standardized in 1986, long before the W3C was founded in 1994. The W3C avoids the wording *Standard* and calls it *Recommendation*, such that the first W3C Recommendation of XML version 1.0 was published in 1998. This W3C Recommendation contains the description of the tree data model of XML, its textual representation, and a first language to express schemas of XML, the DTD. The W3C released the more expressive language XML Schema (Peterson et al. 2009) for schemas of XML data 6 years later in 2004. The basic XML query language, XPath (W3C 2007b), was released in 1999. The name of XPath derives from its basic concept, the path expression, with which the user can hierarchically address the nodes of the XML data. XSLT (W3C 2007a), a language for expressing transformations of XML data, which has been released in the same year as XPath, embeds XPath as well as XQuery (W3C 2007c), which was developed as query language on top of XPath and first occurred in 2002 as working draft. XQuery was finally released in its version 1.0 in 2007 as W3C Recommendation together with XPath 2.0 and XSLT 2.0, which influenced one another.

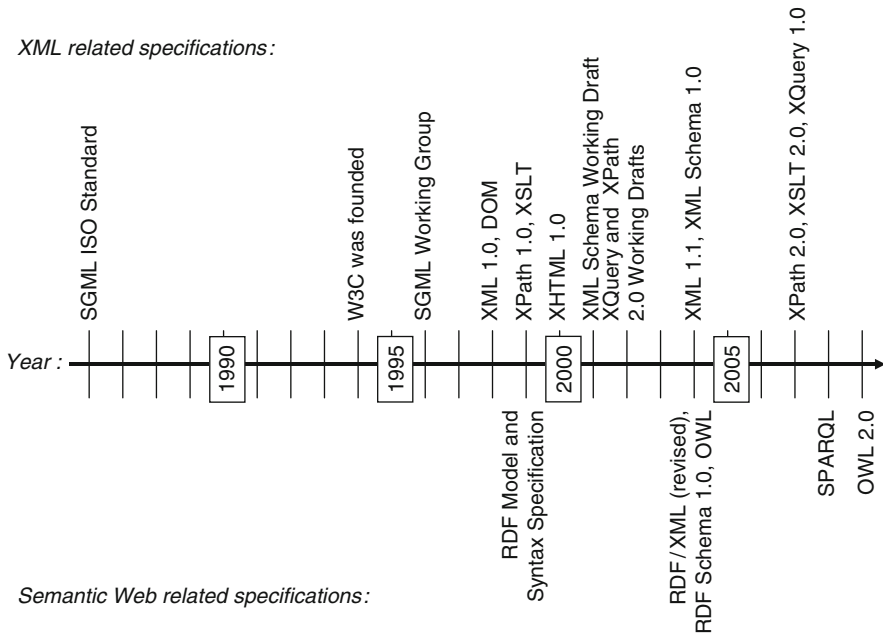


Fig. 12.1 Time line of XML-related and Semantic Web-related specifications

The basic data format RDF (Beckett 2004) of the Semantic Web firstly occurred in the year 1999. RDF data consist of triples, which express labelled directed relationships and which together describe a graph. An ontology is a specification of a conceptualization, which contain the concepts and relationships of the considered domain. Semantic Web ontologies are usually expressed in RDF Schema (Brickley and Guha 2004) or in the Web Ontology Language (OWL) (Dean and Schreiber 2004; Motik et al. 2009). Both ontology languages, RDF Schema and OWL, became W3C Recommendations in 2004, and OWL version 2.0 (Motik et al. 2009) in 2009. While RDF Schema is powerful enough for simple ontologies, OWL is intended for large and complex ontologies. Considering the technical view, ontologies describe constraints on the Semantic Web data, and new facts can be inferred based on a given ontology. Especially, the inference of new facts distinguishes ontologies from schemas as known from databases. The W3C recently published SPARQL (Prud'hommeaux and Seaborne 2008) for querying RDF data as W3C Recommendation in 2008, and works on SPARQL version 1.1. Furthermore, a W3C working group plans to specify a rule language called *Rule Interchange Format* (RIF) (Boley and Kifer 2009; Boley et al. 2009; Sainte Marie et al. 2009) with which one can express arbitrary rules.

Especially, the XML family, but also the Semantic Web family, contains numerous specifications, where we focus on the most important ones specifying the data models, query languages, schemas, and ontologies.

12.2 Concepts and Visions

Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. Today, we use more and more of XML's capability of labeling the information content of diverse data sources including structured and semistructured documents, relational databases, and object repositories.

The Semantic Web aims to provide common formats for integration and combination of data drawn from diverse sources, where the traditional XML-Web mainly concentrated on the interchange of documents. Furthermore, the Semantic Web aims to provide a language for recording how the data relate to real-world objects. One vision of the Semantic Web is that a person, or a machine, can start off in one database and then move through an unending set of databases that are connected by being about the same thing.

The Semantic Web requires that we all or at least a big group agree on the semantics of machine-processable symbols, such that the visions of the Semantic Web work. Thus, for making these visions of the Semantic Web work, one question is if such agreements of a maximized big group of people are realistic and how they can be achieved. Furthermore, how much data already have an agreed semantics and how much more accurate can data be processed by additionally considering the agreed semantics? When providing schemas for XML data, such that valid XML data are an agreed representation of certain objects, which are maybe objects of the real world, and are processed in defined way, XML users also agree on the semantics of machine-processable symbols. Therefore, other questions are how much and in which extensiveness such agreements have been already made with other traditional technologies and which technologies – Semantic Web technologies or traditional technologies – are more suitable for such agreements and for the development of applications, which work with the data with the agreed semantics.

In order to rate the technologies especially regarding the last question, we have to get an overview over the data models, the schema, and the ontology and query languages.

12.3 Data Models

The XPath and XQuery data model (Fernández and Robie 2001) is defined as follows:

Definition 1 (Data Model of XPath and XQuery). *An XML document is a tree of nodes. The kinds of nodes are document, element, attribute, text, namespace, processing instruction, and comment. Every node has a unique node identity that distinguishes it from other nodes. In addition to nodes, the data model allows atomic values, which are single values that correspond to the simple types defined in (Peterson et al. 2009), such as strings, Booleans, decimals, integers, floats, doubles,*

and dates. The first node in any document is the document node, which contains the entire document. Element nodes, comment nodes, and processing instruction nodes occur in the order in which they are found in the textual representation of the XML document. Element nodes occur before their children – the element nodes, text nodes, comment nodes, and processing instructions, which they contain. Attribute nodes and namespace nodes are not considered as children of an element.

XML has a textual representation, which is readable by humans and computers, as well as an application programming interface (API), called DOM (W3C 2004a), for an in memory access to the XML data.

The data model of RDF has been already described in Chap. 2.

There are different ways to represent RDF data, for example, RDF triples (Grant and Beckett 2004), N3 (Berners-Lee 1998), Turtle (Beckett 2006), or RDF/XML (Beckett 2004), which uses XML to encode RDF data.

Figure 12.2 contains a comparison of the XML and Semantic Web data models.

XML data represent a tree of information. RDF data consist of triple data, which expresses a graph. Nevertheless, XML data can contain information to represent graphs and RDF data can express trees.

12.4 Schema and Ontology Languages

W3C's DTDs (W3C 2004b) and XML Schema [see (Peterson et al. 2009)] are two widely used and supported XML schema languages. DTDs are compact and highly readable and can be defined inline. However, DTDs are primarily structural in nature. DTDs have limited support for defining the type of data, and do not have ability to specify specific and precise data types above and beyond character data.

As well as imposing the constraints of structure and semantics on XML documents as DTDs do, the XML Schema language provides powerful capabilities for specifying more concrete data types on elements and attributes, most of which are not expressible in DTDs. The XML Schema language provides a large number of built-in simple types and allows deriving new types for values of elements and attributes, which are only specified to be character data in DTDs.

Since XML Schema can express more restrictions than a DTD, a DTD can be easily transformed into an XML Schema representation, but in general, an XML Schema definition cannot be transformed into a DTD without losing information.

Furthermore, the schemas written in the XML Schema language are XML documents, but the syntax of DTDs is completely different. Therefore, XML Schema can leverage various tools that have been built around XML, but DTDs cannot.

An ontology is the specification of a conceptualization, which contains the concepts and relationships of the considered domain. Semantic Web ontologies are usually expressed in RDF Schema (Brickley and Guha 2004) or in the Web Ontology Language (OWL) (Dean and Schreiber 2004; Motik et al. 2009). While RDF Schema is powerful enough for simple ontologies, OWL is intended for large and complex ontologies. Chapter 2 provides a further introduction to ontology

	XML (XPath and XQuery data model [Fernández and Robie, 2001])	Semantic Web
Underlying structure	Ordered tree	Graph with directed and labelled relationships
Type of nodes	document, element, attribute, text, namespace, processing-instruction and comment	RDF URI reference, blank node or literal (plain, typed or language-tagged)
Values	Single values that correspond to the simple types defined in (W3C, 2001), such as <i>strings</i> , <i>Booleans</i> , <i>decimals</i> , <i>integers</i> , <i>floats</i> , <i>doubles</i> and <i>dates</i>	Plain literals having optionally a language tag, or a typed literal having additionally a datatype URI being a RDF URI reference
Order	The first node in any document is the document node, which contains the entire document. Element nodes, comment nodes, and processing instruction nodes occur in the order in which they are found in the textual representation of the XML document. Element nodes occur before their children –the element nodes, text nodes, comment nodes, and processing instructions, which they contain. Attribute nodes and namespace nodes are not considered as children of an element.	No order between the triples of the RDF data.
Representations	Textual representation, DOM (API)	RDF/XML, RDF Triplets, NTriplets, Turtle

Fig. 12.2 Comparison of the XML and Semantic Web data models

languages. Considering the technical view, ontologies describe constraints on the Semantic Web data, and new facts can be inferred based on a given ontology. Especially, the inference of new facts distinguishes ontologies from schema as known from databases.

12.5 Query Languages

We first compare the XPath language with the SPARQL language, as XPath is embedded in XQuery and XSLT. Later, we also compare XQuery and XSLT with SPARQL.

Tree-based queries can be easily expressed in the tree query languages XPath, XQuery, and XSLT in comparison to the graph query language SPARQL.

For example, SPARQL does not allow computing all descendant nodes of a node like XPath does. However, the formulation of joins in graphs is easier in SPARQL than in XPath.

Both XPath and SPARQL support complex queries and support the usage of variables, constraining the result of queries (see predicates [...] in XPath and **FILTER** expressions in SPARQL) and joins using variables. XPath and SPARQL support iterating through an input dataset (see **for** clauses in XPath and triple patterns in SPARQL). Furthermore, XPath and SPARQL have a rich set of built-in functions, some of which are equivalent (see e.g., function **fn:matches** in XPath and the equivalent function **regex** in SPARQL). Both XPath and SPARQL do not support user-defined functions. Both languages support conditional results (see e.g., **if-then-else** expressions in XPath and **OPTIONAL** patterns in SPARQL) and support nesting of their expressions and statements. The supported datatypes in XPath queries are the datatypes of XML Schema, which are supported in SPARQL, too.

XPath supports mechanisms to determine transitive closures by built-in mechanisms (e.g., using the **descendant** axis), but SPARQL does not support the determination of transitive closures.

Both XPath and SPARQL define built-in functions, which do not have a corresponding built-in function in the other language (see e.g., **fn:replace** and simple aggregates such as **fn:count** and **fn:max** in XPath, and **isIRI** and **isBound** in SPARQL). Cast operations in XPath queries of data of a not castable datatype lead to an error, which stops the evaluation of the XPath query, while casting in SPARQL queries within filter expressions constraints the input data.

XPath expressions return a single or a sequence of atomic values, which are single values that correspond to the simple types defined in (W3C 2001), such as strings, Booleans, decimals, integers, floats, doubles and dates, or a non-nesting, un-typed sequence of nodes, which are ordered according to the document order, whereas the evaluation of SPARQL queries returns a set of bindings of variables.

XQuery and XSLT have the same expressive power, such that translation schemes between these languages exist (see e.g., Klein et al. 2005; Bettentrupp et al. 2006; Groppe et al. 2009c).

XQuery, XSLT as well as SPARQL, can generate the data in the format of their input data; that is, XQuery and XSLT can generate XML fragments and SPARQL can generate RDF data (by using CONSTRUCT queries).

XQuery and XSLT support user-defined functions formulated as XQuery functions and as XSLT templates, but SPARQL does not support user-defined functions formulated in SPARQL. XQuery and XSLT restricted by using intermediate variables support nesting of expressions with full generality, but SPARQL does not support, for example, querying the result of a SPARQL subquery formulated in one SPARQL query. XQuery and XSLT support the determination of transitive closures by, for example, defining and using a recursive function, but SPARQL does not support the determination of the transitive closure.

The good news is that XPath queries can be translated into SPARQL queries (see Droop et al. 2007, 2008, 2009) and SPARQL queries can be translated into XQuery queries and XSLT stylesheets (Groppe et al. 2008a, b).

Figure 12.3 summarizes the comparison between XPath, XQuery, XSLT, and SPARQL.

Support	XPath 2.0	XQuery/XSLT	SPARQL
Type	Tree query language	Tree query language	Graph query language
Variables	Yes	Yes	Yes
Constraints	Yes (Predicates)	Yes (Predicates/ where-clause (XQuery))	Yes (Filter-clause)
Joins	Yes (using variables)	Yes (using variables)	Yes (using variables)
Iteration through input data set	Yes (for-clause)	Yes (for-clause/for-each XSLT instruction)	Yes (triple patterns)
Built-in functions	Yes	Yes	Yes
User-defined functions	No	Yes (functions in XQuery and templates in XSLT)	No
Conditional results	Yes (if-then-else expression)	Yes (if-then-else expression /if XSLT instruction)	Yes (Optional patterns)
XML Schema data types	Yes	Yes	Yes
Nesting of expressions	Yes	Yes (in XSLT intermediate results have often to be stored in variables)	Restricted (e.g. sub-queries planned for SPARQL 1.1)
descendant nodes	Yes	Yes	No
Determination of transitive closure	Built-in mechanisms to retrieve descendant/ ancestor nodes	Yes (recursive functions and templates)	No
Cast errors	Stop evaluation	Stop evaluation	Constraints input data
Type of query result	single or a sequence of atomic values, or a non-nesting, un-typed sequence of nodes, which are ordered according to the document order	Like the query result type of XPath and additionally XML fragments	a set of bindings of variables, a Boolean value or an RDF graph
Updates	No	No (Proprietary extension of XQuery for updates available)	No (planned for SPARQL 1.1)
Translations	XPath into SPARQL (Droop et al. 2009)	XQuery/XSLT can be translated into each other (Groppe et al. 2009c), but not into SPARQL	SPARQL into XQuery/XSLT (Groppe et al. 2008)

Fig. 12.3 Comparison of XPath, XQuery, XSLT, and SPARQL features

12.6 Embedding SPARQL into XQuery/XSLT

The tree-based languages XQuery and XSLT for XML are widely supported. We propose to embed SPARQL subqueries into XQuery/XSLT, such that XQuery and XSLT benefit from the graph query language constructs of SPARQL, and SPARQL benefits from features of XQuery/XSLT, which SPARQL does not support. The embedding enables XQuery/XSLT tools to handle at the same time XML queries and SPARQL subqueries, and XML and RDF data.

Many commercial as well as freely available products support the evaluation of XQuery expressions or XSLT stylesheets, but do not support the SPARQL language. Examples for products with XQuery support include Tamino XML Server (Software AG 2007), Microsoft SQL Server 2005 Express (Microsoft 2007), Saxon (Kay 2010), and Qizx (xmlmind 2010). Examples for products with XSLT support include BizTalk (Microsoft 2009), Cocoon (Apache 2009), and Saxon (Kay 2010). Integration of RDF data into XML data and embedding of SPARQL queries into XQuery queries and XSLT stylesheets can make RDF data and SPARQL available for these products. Furthermore, the proposed embedding enables users to work in parallel with XML data and RDF data, and with XQuery queries, XSLT stylesheets, and SPARQL queries; that is RDF data are integrated in XML data and the result of SPARQL subqueries can be used in XQuery/XSLT for further processing. As most XQuery/XSLT tools do not allow calling an external SPARQL evaluator from XQuery/XSLT, we propose to translate embedded SPARQL queries into XQuery/XSLT subexpressions.

Graph queries can be easily expressed in the graph query language SPARQL in comparison to the tree-based query and transformation languages XQuery and XSLT. An embedding of SPARQL into XQuery/XSLT allows the easy formulation of tree queries/transformations and graph queries in one query. Furthermore, SPARQL does not support many language constructs such as computation of the transitive closure and user-defined functions, which are supported by the proposed embedding. Thus, the host languages XQuery and XSLT benefit from the embedded language SPARQL and the embedded language SPARQL benefits from its host languages XQuery and XSLT.

The contributions of this section are as follows:

- The embedding of SPARQL into XQuery/XSLT, which enables users to benefit from both graph and tree language constructs,
- The translation from the operator tree into XQuery/XSLT, and
- An experimental evaluation of the embedding.

12.6.1 Embedded SPARQL

In this section, we introduce language constructs for embedding SPARQL in XQuery/XSLT. The RDF data of Fig. 12.4 contain a FOAF (Brickley and Miller 2007)

Fig. 12.4 RDF data

```
(1) @prefix foaf: <http://xmlns.com/foaf/0.1/> .
(2) _:a foaf:name "Alice".
(3) _:a foaf:mbox <mailto:alice@work.example> .
(4) _:b foaf:name "Bob" .
(5) _:b foaf:mbox <mailto:bob@home.example> .
```

```
(1) declare namespace foaf = "http://xmlns.com/foaf/0.1/";
(2) <results>{ for ($n, $m) in
(3)           SELECT ?name ?mbox
(4)           WHERE { ?x foaf:name ?name .
(5)             ?x foaf:mbox ?mbox .
(6)           FILTER regex(str(?mbox),"@work.example") }
(7) return   <result>
(8)           <name>{$n}</name>
(9)           <mbox>{$m}</mbox>
(10)        </result>}
(11) <results>
```

Fig. 12.5 Embedded SPARQL query Q in an XQuery query

document for representing social networks with four triples, the first of which (line (2)) associates `_:a` (the subject) with `foaf:name` (the predicate) and "Alice" (the object). In this chapter, we do not resolve the prefix `foaf`, which is defined in line (1), due to simplicity of presentation.

Lines (3–6) in Fig. 12.5 and lines (5–10) in Fig. 12.6 show examples of an embedded SPARQL (Prud'hommeaux and Seaborne 2008) query Q in XQuery (W3C 2007c) and in XSLT (W3C 2007a). The embedded SPARQL query Q returns the names and email addresses of those people, the email addresses of which contain "@work.example". The result of a SPARQL query is a *bag* (also called *multiset*), that is, its unordered elements can appear more than once. The result of Q is the bag $\langle \{(\text{name}, \text{"Alice"}), (\text{mbox}, \langle \text{mailto:alice@work.example} \rangle)\} \rangle$ of environments consisting of bindings of variables; for example, the value "Alice" is bound to the variable `name`.

The proposed extension for $(\$var_1, \$var_2, \dots, \$var_n)$ in S of XQuery and the proposed extension `<xsl:for-SPARQL var="\$var_1, \$var_2, \dots, \$var_n">S</xsl:for-SPARQL>` of XSLT, where S is a SPARQL SELECT subquery, iterate through all environments of the result of S. In each iteration, the values of bound variables of the current environment are mapped to the XQuery/XSLT variables $\$var_1, \dots, \var_n , which can be afterward used in normal XQuery/XSLT instructions. For example, Fig. 12.7 presents the overall result of the XQuery query of Fig. 12.5 and of the XSLT stylesheet of Fig. 12.6, both of which with embedded SPARQL subquery.

```

(1) <xsl:stylesheet xmlns:foaf = "http://xmlns.com/foaf/0.1/">
(2)   <xsl:template match = "/">
(3)     <results>
(4)       <xsl:for -SPARQL var = "$n $m">
(5)         <xsl:SPARQL>
(6)           SELECT ?name ?mbox
(7)           WHERE   { ?x foaf:name ?name .
(8)                   ?x foaf:mbox ?mbox .
(9)                   FILTER regex(str(?mbox), "@work.example") }
(10)        </xsl:SPARQL>
(11)      <result>
(12)        <name><xsl:value-of select = "$n"/></name>
(13)        <mbox><xsl:value-of select = "$m"/></mbox>
(14)      </result>
(15)    </xsl:for-SPARQL>
(16)  </results>
(17) </xsl:template>
(18) </xsl:stylesheet>

```

Fig. 12.6 Embedded SPARQL query Q in an XSLT stylesheet

```

<results>
  <result>
    <name>"Alice"</name>
    <mbox>mailto:alice@work.example</mbox>
  </result>
</results>

```

Fig. 12.7 Result of Fig. 12.5/Fig. 12.6 with Fig. 12.4 as input

Additionally, we propose the extension $\text{let } (\$var_1, \$var_2, \dots, \$var_n) := S$ of XQuery and the extension $\text{<xsl:result-SPARQL var="}\$var_1, \$var_2, \dots, \$var_n\text{">S</xsl:result>}$ of XSLT, where S is a SPARQL SELECT subquery, which map a whole result of S to the variables $\$var_1, \dots, \var_n , each of which stores sequences of values, such that $\$var_1[i], \dots, \$var_n[i]$ represents the i th result of S .

SPARQL ASK queries return a Boolean value, such that an embedding of ASK subqueries in Boolean expressions of XQuery and XSLT is natural, which we omit here due to no further scientific insights.

12.6.2 Translation Process

In this section, we describe the translation process, which consists of the integration of RDF data into XML data and the translation from the operator tree into XQuery/XSLT subexpressions.

12.6.2.1 Integration of RDF Data into XML

The translated embedded SPARQL subqueries formulated in XQuery/XSLT require a simple form of the RDF data in XML format, such that each triple can be easily accessed by XQuery and XSLT language constructs. Thus, we iterate through all triples and translate each triple (*sv*, *pv*, *ov*) with optional datatypes *sd*, *pd*, and *od* into an XML subtree.

```
<entry>
  <s>sv</s>
  <s_datatype>sd</s_datatype>
  <p>pv</p>
  <p_datatype>pd</p_datatype>
  <o>ov</o>
  <o_datatype>od</o_datatype>
</entry>
```

If *sd*, *pd*, or *od* are not given, then we do not generate the corresponding element `<s_datatype>`, `<p_datatype>`, or `<o_datatype>`, respectively. All these `<entry>` elements are integrated in an already existing input XML document. In the example of Fig. 12.8, which contains the translated RDF data of Fig. 12.4, we insert them as children under a `<translatedData>` element.

12.6.2.2 Physical Operators Formulated in XQuery/XSLT

In this section, we present algorithms of the operators formulated in XQuery and XSLT, as most XQuery/XSLT tools do not allow calling an external SPARQL evaluator from XQuery/XSLT.

We use transformation rules to represent the translation from the operator tree of the embedded SPARQL query into XQuery/XSLT. The transformation rule $T(\text{Op}, \text{operand}) \Rightarrow r$ or $T(\text{Op}, \text{operand}_1, \dots, \text{operand}_n) \Rightarrow r$, respectively, replaces an operator *Op* with subtree operand (or with operands *operand*₁, ..., *operand*_{*n*}, respectively) by the right-hand side *r*. *r* consists of XQuery/XSLT expressions, which occur in the translation as in *r*, in *italic*, conditional processing instructions, and calls of other transformation rules and helper functions.

```

<translatedData>
  <entry>
    <s>a</s>
    <p>http://xmlns.com/foaf/0.1/mbox</p>
    <o>mailto:alice@work.example</o>
  </entry>
  <entry>
    <s>a</s>
    <p>http://xmlns.com/foaf/0.1/name</p>
    <o>Alice</o>
  </entry>
  <entry>
    <s>b</s>
    <p>http://xmlns.com/foaf/0.1/mbox</p>
    <o>mailto:bob@home.example</o>
  </entry>
  <entry>
    <s>b</s>
    <p>http://xmlns.com/foaf/0.1/name</p>
    <o>Bob</o>
  </entry>
</translatedData>

```

Fig. 12.8 Simple XML form of the RDF data of Fig. 12.4

The result of a call $T(\text{Op}, \text{operand})$, where Op is the root operator of the operator tree and operand its subtree, is the translation of an embedded SPARQL query into an XQuery subquery or XSLT subexpression, respectively, which replaces the embedded SPARQL query in the original XQuery query/XSLT stylesheet.

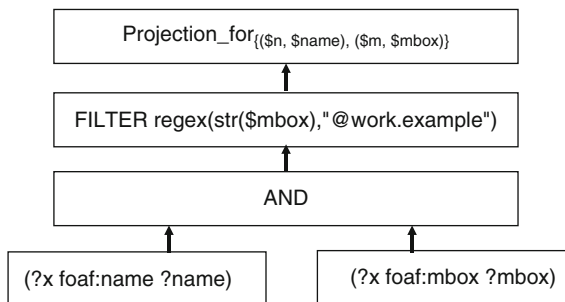
The call of a transformation rule $T(\text{Op})$, where Op is an operator, is equivalent to a call of the transformation rule $T(\text{Op}, \text{operand}_1, \dots, \text{operand}_n)$, where $\text{operand}_1, \dots, \text{operand}_n$ are the operands of Op . We use the same short notation for helper functions.

Many transformation rules and helper functions are the same for XQuery and XSLT. Whenever they are different, we mark them with a subscript “XQuery” or “XSLT,” respectively; that is $T_{\text{XQuery}}(\text{Op}, \text{operand}) \Rightarrow r$ represents a transformation rule for XQuery, $T_{\text{XSLT}}(\text{Op}, \text{operand}) \Rightarrow r$ for XSLT, and $T(\text{Op}, \text{operand}) \Rightarrow r$ for both translations, for XQuery and XSLT.

Intermediate variables generated by the translation and SPARQL variables may conflict with already used variables of the outer host XQuery/XSLT expression because of same names. Therefore, we consistently rename those variables. We do not present the variable renaming algorithm here due to simplicity of presentation.

We extend our SPARQL algebra as discussed before by two new operators, which represent the functionality of the operators for embedding SPARQL into XQuery/XSLT:

Fig. 12.9 Operator graph of the embedded SPARQL query in Fig. 12.5 and in Fig. 12.6



1. The projection operator $\text{Projection_for}_{\{(x_1, s_1), \dots, (x_n, s_n)\}}$ represents the language extension for $(\$var_1, \$var_2, \dots, \$var_n)$ in ... of XQuery and the extension `<xsl:for-SPARQL var="$var_1, $var_2, \dots, $var_n"> ... </xsl:for-SPARQL>` of XSLT. For example, $\text{Projection_for}_{\{(n, \text{name}), (m, \text{mbox})\}}$ represents the extension of XQuery/XSLT and the `SELECT` clause in line (2) of Fig. 12.5 and line (5) of Fig. 12.6. The semantics of the projection operator $\text{Projection_for}_{\{(x_1, s_1), \dots, (x_n, s_n)\}}$ defines a loop for iterating its succeeding operators with the variables x_1, \dots, x_n of the host XQuery/XSLT expression assigned with the bound values of the SPARQL variables s_1, \dots, s_n of the embedded SPARQL query result.
2. The projection operator $\text{Projection_let}_{\{(x_1, s_1), \dots, (x_n, s_n)\}}$ represents the result of the language extensions `let ($var_1, $var_2, \dots, $var_n) := ...` and `<xsl:result-SPARQL var="$var_1, $var_2, \dots, $var_n"> ... </xsl:result>` of the host languages. The projection operator $\text{Projection_let}_{\{(x_1, s_1), \dots, (x_n, s_n)\}}$ assigns the variables x_1, \dots, x_n of the host XQuery/XSLT expression with the sequence of bound values of the SPARQL variables s_1, \dots, s_n of the embedded SPARQL query result.

With this extended SPARQL algebra, the embedded SPARQL query in Fig. 12.5 and in Fig. 12.6 can be represented as operator graph as depicted in Fig. 12.9.

XQuery

See Fig. 12.10 for the result of the translation of the embedded SPARQL query of Fig. 12.5 into XQuery.

The following transformation rule for projection operators is the entry point for translating the operator tree of a SPARQL query. The translated XQuery query contains the declaration of a variable addressing all translated RDF triples. First, the subtree of the projection operator is translated and then the variables to which it is projected and their bound values are set. If there are optional variables, which are stored in $\$Optional_1$ to $\$Optional_n$, these optional variables are considered, too. The function `boundVariablesCurrentScope.contains` is defined to return true for those variables, which are not optional variables, and otherwise false. The function `getUniqueID()` returns unique identifiers and the function `getLastID()` (`getPrelastID()`, respectively) returns the identifier of the last call (pre-last call respectively) of `getUniqueID()`.

```
TXQuery(Projection_for{(x1, s1), ..., (xn, sn)}, s) => let $__entry__ := /translatedData/entry;
                                     T(s)pXQuery(x1, s1)...pXQuery(xn,sn)
```

```
TXQuery(Projection_let{(x1, s1), ..., (xn, sn)}, s) =>
let $__entry__ := /translatedData/entry;
let $_i_getUniqueID(): = T(s) return <value name = "x1">pXQuery(x1, s1)</value>...
                                     <value name = "xn">pXQuery(xn, sn)</value>
let $x1 := $_i_getLastID()[@name = "x1"/text()]...
let $xn := $_i_getLastID()[@name = "xn"/text()]
```

```
PXQuery(xi,si) => if (boundVariablesCurrentScope.contains(vi)) then let $xi := $si
else let $xi := for $_i_getUniqueID() in
    $Optional1/binding/var[. = "vi"/following-sibling::value, ...,
    $Optionaln/binding/var[. = "vi"/following-sibling::value
    return $_i_getLastID()
```

The transformation rule for a triple pattern operator (p1 p2 p3) generates XQuery instructions, which check for each triple in the input RDF data, whether or not the triple pattern matches the considered triple (in the where helper function) by also considering already bound variables. The generated XQuery instructions also bind variables in the triple pattern to their corresponding values of the considered triple. The function boundVariables.contains(pi) returns true if the variable pi has been already bound to a value because of the generated XQuery instructions of a previous triple pattern operator. getVarOf((p1 p2 p3)) returns the name of the variable holding the currently considered triple of the triple pattern operator (p1 p2 p3).

```
TXQuery((p1 p2 p3)) => for $_cE_getUniqueID() in $__entry__
    patXQuery(p1, s, _cE_getLastID())
    patXQuery(p2, p, _cE_getLastID())
    patXQuery(p3, o, _cE_getLastID())
```

```
patXQuery(pi, pos, varOfTriple) =>
if(pi is variable and !boundVariables.contains(pi))
then let $pi := $varOfTriple /pos let $pi_datatype := $varOfTriple /pos_datatype
```

```
where((p1p2 p3)) => wherePattern(p1, s, getVarOf((p1 p2 p3))) and
    wherePattern(p2, p, getVarOf((p1 p2 p3))) and
    wherePattern(p3, o, getVarOf((p1 p2 p3)))
```

```
wherePattern (pi, pos, varOfTriple) =>
if(pi is a variable)
then { if(boundVariables.contains(pi)) then varOfTriple /pos /text() = $pi}
else varOfTriple /pos /text() = pi.getString()
```


The translation rules for filter expressions add a translated XPath expression to the constraints to be checked in the outer where clause of the translated XQuery expression.

$T(\text{FILTER } R, s) \Rightarrow T(s)$
 $\text{where}(\text{FILTER } R, s) \Rightarrow R \text{ and } \text{where}(s)$

The translation of the UNION operator and its operands is analogous to the translation of the optional part of Optional.

$T(\text{UNION}, s_1, \dots, s_n) \Rightarrow \text{optional1}(s_1) \dots \text{optional1}(s_n)$
 $\text{where}(\text{UNION}, s_1, \dots, s_n) \Rightarrow$

XSLT

We present only those transformation rules (see Fig. 12.11), which differ from the transformation rules for XQuery, in the following paragraphs. As analogous remarks in comparison to the XQuery transformation rules apply, we present the transformation rules for XSLT without remarks. The function `close(s)` returns `</xsl:if>` for each open `<xsl:if>` and `</xsl:for-each>` for each open `<xsl:for-each>` according to their nesting of the translations of the operator `s` in order to generate correct XSLT stylesheets. For the $T_{\text{XSLT}}(\text{Projection_for}_V, s)$ operator, we have to replace the closing tag `</xsl:for-SPARQL>` with the result of `close(s)`. See Fig. 12.12 for the result of the translation of the embedded SPARQL query of Fig. 12.6 into XSLT.

```
(1) declare namespace foaf = "http://xmlns.com/foaf/0.1/";
(2) <results>{ let $__entry__ := /translatedData/entry
(3)           for $_cE_0 in $__entry__
(4)           let $x := $_cE_0/s/text()
(5)           let $x_datatype := $_cE_0/s_datatype/text()
(6)           let $name := $_cE_0/o/text()
(7)           let $name_datatype := $_cE_0/o_datatype/text()
(8)           for $_cE_1 in $__entry__
(9)           let $mbox := $_cE_1/o/text()
(10)          let $mbox_datatype := $_cE_1/o_datatype/text()
(11)          let $n = $name $m := $mbox
(12)          where   matches($mbox, "@work.example") and
(13)                  $_cE_0[p/text() = 'http://xmlns.com/foaf/0.1/name'] and
(14)                  $_cE_1[s/text() = $x and
(15)                  $_cE_1[p/text() = 'http://xmlns.com/foaf/0.1/mbox']
(16)          return <result>
(17)                  <name>{$n}</name>
(18)                  <mbox>{$m}</mbox>
(19)          </result>}
(20) </results>
```

Fig. 12.10 Result of translating Fig. 12.5

12.6.3 Experimental Analysis

We present the average of ten execution times of the original SPARQL queries of the DAWG test cases (Feigenbaum 2008), of the data and query translation, and of the processing of the translated queries (see Figs. 12.13–12.20). The DAWG test cases consist of a set of queries covering many aspects of the SPARQL language.

The test system uses an Intel Pentium 4 processor with 2.66 GHz, 1 GB main memory, Windows XP Professional 2002, and Java 1.5. We use Jena (Wilkinson et al. 2003) since it supports the current SPARQL version (Prud'hommeaux and Seaborne 2008), which is not fully supported by many SPARQL processing engines. We choose the widely used Saxon (Kay 2010) XQuery evaluator, and Qizx (xmlmind 2010). Xalan and Saxon (Kay 2010) are widely used Java XSLT processors, where only Saxon supports XSLT 2.0 (W3C 2007a), which is a requirement of our translation.

We exclude those queries for the experiments, which cannot be translated by our prototype for the following reasons:

- The queries contain not supported built-in functions such as LANG, LANG-MATCHES, sameTerm, isIRI, isURI, isBLANK, and isLITERAL, which do not have equivalent XQuery/XSLT built-in functions.

```

TXSLT(Projection_for{(x1, s1), ..., (xn, sn)}, s) =>
  <xsl:variable name = '___entry___' select = '/translatedData/entry' />
  T(s)
  PXSLT(x1,s1)...PXSLT(xn,sn)

TXSLT(Projection_let{(x1, s1), ..., (xn, sn)}, s) =>
  <xsl:variable name = '___entry___' select = '/translatedData/entry' />
  <xsl:variable name = '_i_getUniqueID()'>
    T(s)
    <value name = "x1">PXSLT(x1, s1)</value>...
    <value name = "xn">PXSLT(xn, sn)</value>
    close(s)
  </xsl:variable>
  <xsl:variable name = 'x1' select = $_i_getLastID() [ @name = "x1" ] / text() />...
  <xsl:variable name = 'xn' select = $_i_getLastID() [ @name = "xn" ] / text() />

PXSLT(xi, si) => if(boundVariablesCurrentScope.contains(vi))
  then <xsl:variable name = 'xi' select = '$si' />
  else <xsl:variable name = 'xi' select = '
    $Optional1 / binding / var[. = "si" ] / following-sibling::value / text(), ...,
    $Optionaln / binding / var[. = "si" ] / following-sibling::value / text() />

```

Fig. 12.11 (continued)

```

TXSLT((p1p2p3)) =>
    <xsl:for-each select = '$__entry__'>
    <xsl variable name = '_cE_getUniqueID()' select = './'>
    patXSLT(p1, s, _cE_getLastID())
    patXSLT(p2, p, _cE_getLastID())
    patXSLT(p3, o, _cE_getLastID())

patXSLT(pi, pos, varOfTriple) =>
    if(pi is variable and !boundVariables.contains(pi))
    then
        <xsl:variable name = 'pi' select = '$varOfTriple/pos' />
        <xsl:variable name = 'pi_datatype' select = '$varOfTriple/pos_datatype' />

TXSLT(AND, s1, ..., sn) => T(s1) ... T(sn)
    If(Join is the outermost from all AND / OPT operators)
    then <xsl:if test = 'where(s1) and... and where(sn)'>

TXSLT(OPT, s1, s2) =>
    T(s1) optional1XQuery(s2);
    If(OPT is the outermost from all AND / OPT operators)
    then <xsl:if test = 'where(s1)'>

optional1XSLT(s) =>
    <xsl:variable name = '_t_getUniqueID()'>
    T(s)
    <xsl:if test = 'where(s)'>
    <binding>
    <var> boundVariablesOf(s).get(1) </var>
    <value><xsl:value-of select = '$boundVariablesOf(s).get(1)' /></value>
    </binding>...
    <binding>
    <var> boundVariablesOf(s).getLast() </var>
    <value><xsl:value-of select = '$boundVariablesOf(s).getLast()' /></value>
    </binding>
    </xsl:if>
    close(s)
    </xsl:variable>
    <xsl:variable name = '__optional__getUniqueID()'>
    <xsl:choose>
    <xsl:when test = 'not($_t_getPrelastID())'>
    <noBinding> noBinding(s) </noBinding>
    </xsl:when>
    <xsl:otherwise>
    <xsl:copy-of select = '$_t_getPrelastID()' />
    </xsl:otherwise>
    </xsl:choose>
    </xsl:variable>

```

Fig. 12.11 Transformation rules for XSLT


```

<xsl:stylesheet xmlns:foaf = "http://xmlns.com/foaf/0.1/">
  <xsl:template match = "/">
    <results>
      <xsl:variable name = '__entry__' select = '/translatedData/entry'/>
      <xsl:for-each select = '$__entry__'>
        <xsl:variable name = '_cE_0' select = './>
        <xsl:variable name = 'x' select = '$_cE_0/s'/>
        <xsl:variable name = 'x_datatype' select = '$_cE_0/s_datatype'/>
        <xsl:variable name = 'name' select = '$_cE_0/o'/>
        <xsl:variable name = 'name_datatype' select = '$_cE_0/o_datatype'/>
        <xsl:for-each select = '$__entry__'>
          <xsl:variable name = '_cE_1' select='./>
          <xsl:variable name = 'mbox' select = '$_cE_1/o'/>
          <xsl:variable name = 'mbox_datatype' select = '$_cE_1/o_datatype'/>
          <xsl:if test = 'matches($mbox, "@work.example") and
                        $_cE_0/p/text() = "http://xmlns.com/foaf/0.1/name" and
                        $_cE_1/s/text() = $x and
                        $_cE_1/p/text() = "http://xmlns.com/foaf/0.1/mbox" '>
            <xsl:variable name = 'n' select = '$name'/>
            <xsl:variable name = 'm' select = '$mbox'/>
            <result>
              <name><xsl:value-of select = "$n"/></name>
              <mbox><xsl:value-of select = "$m"/></mbox>
            </result>
          </xsl:if>
        </xsl:for-each>
      </xsl:for-each>
    </results>
  </xsl:template>
</xsl:stylesheet>

```

Fig. 12.12 Result of translating Fig. 12.6



Fig. 12.13 Execution times of the first part of the DAWG test cases, where we use the filenames of the DAWG queries to label test cases

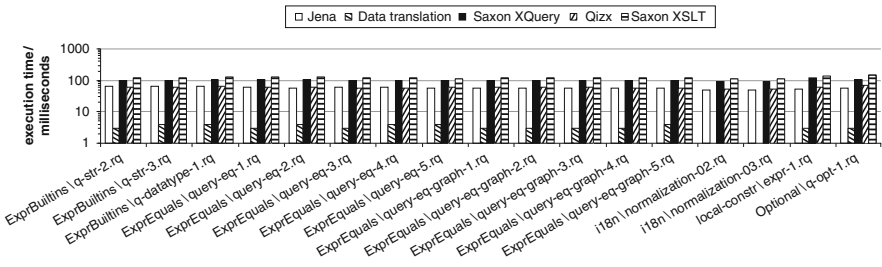


Fig. 12.14 Execution times of the second part of the DAWG test cases, where we use the filenames of the DAWG queries to label test cases

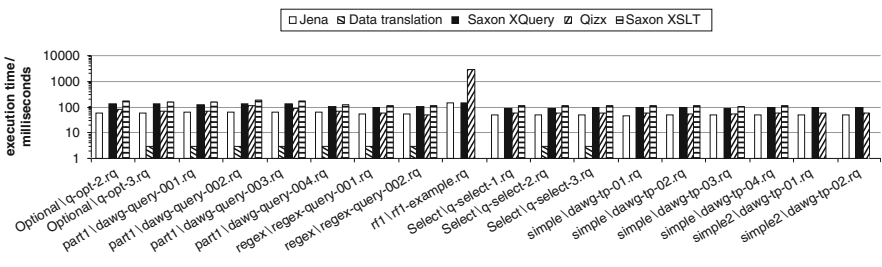


Fig. 12.15 Execution times of the third part of the DWAG test cases, where we use the filenames of the DAWG queries to label test cases



Fig. 12.16 Execution times of the fourth part of the DAWG test cases, where we use the filenames of the DAWG queries to label test cases

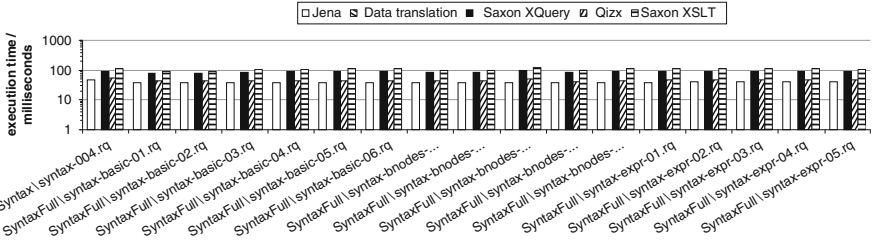


Fig. 12.17 Execution times of the fifth part of the DAWG test cases, where we use the filenames of the DAWG queries to label test cases

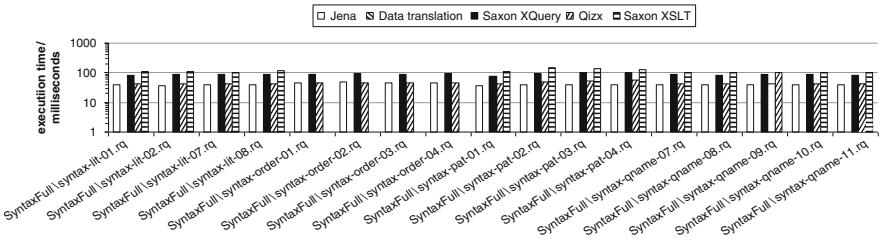


Fig. 12.18 Execution times of the sixth part of the DAWG test cases, where we use the filenames of the DAWG queries to label test cases

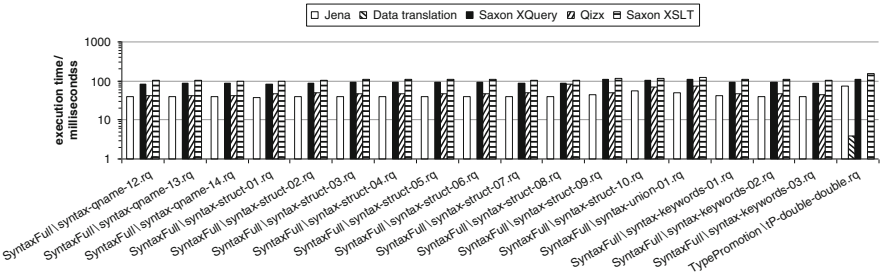


Fig. 12.19 Execution times of the seventh part of the DAWG test cases, where we use the filenames of the DAWG queries to label test cases

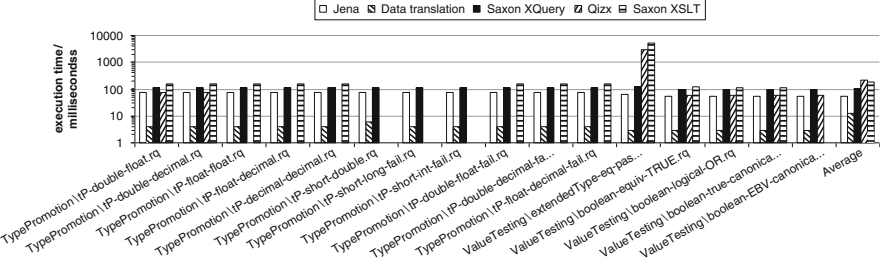


Fig. 12.20 Execution times of the eighth part of the DAWG test cases, where we use the filenames of the DAWG queries to label test cases

- The queries contain FILTER statements with variables of OPTIONAL patterns, in which the considered FILTER statement is not embedded. This is solvable by additional tests of the bindings of variables of OPTIONAL patterns or by transferring the FILTER statement into the corresponding OPTIONAL pattern.
- The queries and data contain other data types than String, Boolean, Decimal, Float, Double, Duration, dateTime, Time, and Date. The used XQuery evaluators and XSLT processors do not support special data types such as nonNegativeInteger.
- For the translation into XSLT: The queries require sorting according to another data type than String, as Saxon XSLT always sorts according to the string representation of the values.

We neglect the support, which can be added by using a function library especially developed for this purpose, of this kind of queries as their support do not demonstrate further main principles.

The average execution time of the original SPARQL queries of these 137 queries is 54.4 ms when using the Jena evaluator. The average execution time for data translation from RDF data to XML data is 12.4 ms; the average execution time for translating the query from SPARQL to XQuery/XSLT is approximately 6 ms. While the executions of the translated queries are for Saxon XQuery 1.86 times, for Qizx 4.06 times, and for Saxon XSLT 3.55 times slower in comparison with using Jena, the absolute average execution times are quite small, 101 ms for Saxon XQuery, 220.6 ms for Qizx, and 184.4 ms for Saxon XSLT.

12.7 Embedding XPath Into SPARQL

In recent years, many RDF storage systems, which support or plan to support SPARQL, have occurred like Jena (Wilkinson et al. 2003). These RDF storage systems do not support XML data and XPath queries, which are currently widely used in applications. Integration of XML data into RDF data and embedding of XPath queries into SPARQL queries can make XML data and XPath available for these products. Furthermore, the proposed embedding enables users to work in parallel with XML data and RDF data, and with XPath queries and SPARQL queries, that is, XML data are integrated in RDF data and the result of XPath subqueries can be used in SPARQL for further processing. As many SPARQL tools do not allow calling an external XPath evaluator from SPARQL, we propose to translate embedded XPath queries into SPARQL subexpressions.

Tree-based queries can be easily expressed in the tree query language XPath in comparison to the graph query language SPARQL. For example, SPARQL does not allow computing all descendant nodes of a node like XPath does. Furthermore, the formulation of joins in graphs is easier in SPARQL than in XPath. An

embedding of XPath into SPARQL allows the easy formulation of tree queries and graph queries in one query. Thus, the host language SPARQL benefits from the embedded language XPath, and the embedded language XPath benefits from its host language SPARQL.

In this section, we propose a translation scheme for XML data into RDF data and XPath queries into SPARQL queries. Furthermore, we present the results of an experimental evaluation of a prototype, which shows that various different XPath queries can be embedded into SPARQL.

12.7.1 Translation of XPath Subqueries Into SPARQL Queries

We propose to embed XPath subqueries in SPARQL queries by binding a SPARQL variable to the result of an XPath subquery by using a BIND(S, E) construct in the WHERE clause of the host SPARQL query, which assigns the resultant nodes of an embedded XPath expression E to a SPARQL variable S. For example, we present a SPARQL query with an embedded XPath subquery in Fig. 12.21, where the titles of all available books in a collection of books from a bookstore, the data of which are stored in the input XML document, are retrieved and represented by the SPARQL variable ?XPath [see line (6)]. Furthermore, the retrieved result of the embedded XPath subquery is compared with the titles in the book collection of the user [see line (9)], which are stored in the input RDF document, and the current places of the books are determined [see line (10)]. The titles of the books available in both the bookstore and the book collection of the user are returned together with the current place of the book [see line (5)].

We first explain how to translate an XPath query into an equivalent SPARQL query. Afterward, we explain how to integrate the translation of an embedded XPath subquery into its host SPARQL query.

The translation process consists of three steps (see Fig. 12.22): (1) the translation of the input data from XML into RDF, (2) the source-to-source translation from the XPath query into the translated SPARQL query, and (3) in the case that the host

```
(1) PREFIX myCollection:
(2)       <http://www.luposdate.org/>
(3) PREFIX xsd:
(4)       <http://www.w3.org/2001/XMLSchema#>
(5) SELECT ?XPath ?place WHERE
(6) { BIND(?XPath,
(7)       /child::bookstore/parent::node()/
(8)       descendant::title/child::text()).
(9)   ?x myCollection:title ?XPath.
(10)  ?x myCollection:place ?place. }
```

Fig. 12.21 Host SPARQL query with embedded XPath query

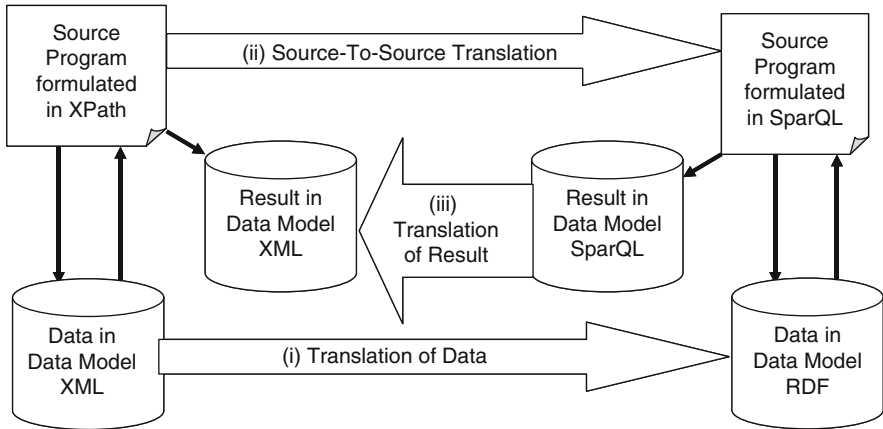


Fig. 12.22 The translation process consisting of three steps

SPARQL query returns the result of an embedded XPath query the translation of the result from the translated SPARQL query into the result according to the XPath and XQuery data model, which is equivalent to the result of the XPath query. We explain each translation step in detail in the following subsections.

12.7.1.1 Translation of Data

The implicit type of each node of the XML tree has to be stored explicitly by a special relationship for translating XML data into RDF data. All implicit relationships of nodes of the XML tree have to be added as explicit relationships in the RDF data. This includes parent–child relationships, attribute and namespace relationships, and next-sibling relationships. As SPARQL does not support the determination of transitive closures, we have to add a numbering scheme to the RDF data in order to support XPath axes, which require the determination of the transitive closure of basic relationships, such as descendant, ancestor, following, and preceding.

We translate the XML data into RDF data by a depth-first traversal of the XML tree and annotate each translated node of the XML data with the relationships `rel:type`, `rel:child`, `rel:attribute`, `rel:namespace`, `rel:name`, `rel:value`, `rel:start`, and `rel:end`.

As an example, see Fig. 12.23 for the original XML data, see Fig. 12.24 for its graphical representation, and see Fig. 12.25 for the translated RDF graph.

We use a relationship `rel:type` in the RDF data in order to explicitly annotate the type of an XML node. The explicit relationship `rel:child` in the translated RDF data expresses a parent–child relationship in the XML tree, `rel:attribute` an attribute relationship, and `rel:namespace` a namespace relationship. The value associated with the relationship `rel:name` contains the name of the XML node; `rel:value` contains the value of the node. The relationships `rel:start` and `rel:end` are computed by a numbering scheme for XML data and their purpose are for the determination of

Fig. 12.23 An example XML document representing a bookstore

```
<bookstore>
  <book category = "CHILDREN">
    <title>Harry Potter</title>
    <author>J. K. Rowling</author>
  </book>
  <book category = "WEB">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
  </book>
</bookstore>
```

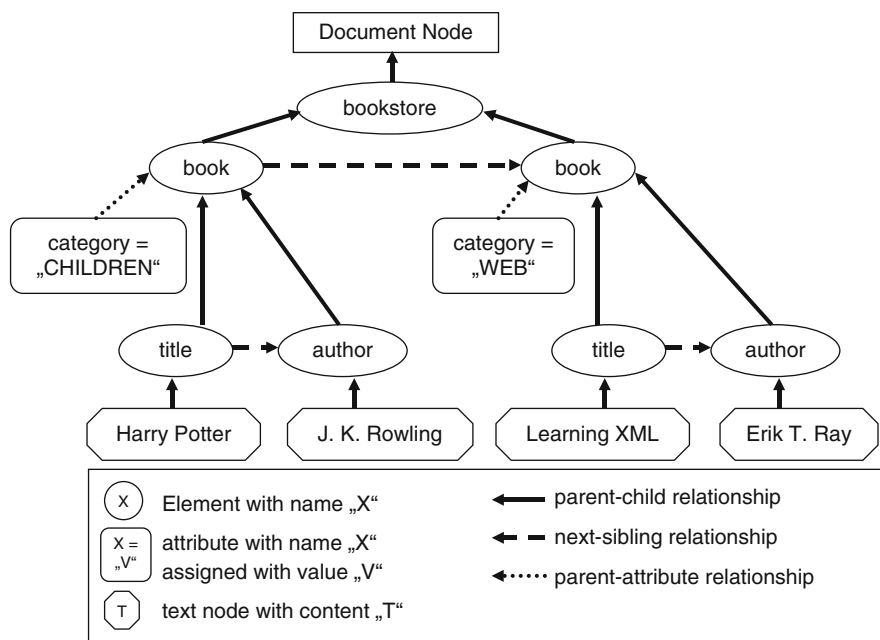


Fig. 12.24 Graphical representation of the XML document of Fig. 12.23

the descendant relationships in SPARQL queries, as SPARQL does not support the determination of transitive closures.

We use a numbering scheme based on the region encoding on elements of the XML tree, which we adapt from Grust et al. (2004): For each element, the values of rel:start and rel:end can be assigned by a depth-first traversal through the XML tree. The value of rel:start of the document tree is 1. The value rel:end of a node v with start value n can be computed by $n + \text{count}(\text{subtree}(v)) + 1$, where the function count returns the number of nodes of the subtree rooted at v . The value of rel:start of the first child of a node v is $v.\text{start} + 1$. The value of rel:start of a non-first child is the value of rel:end of the previous sibling plus 1.

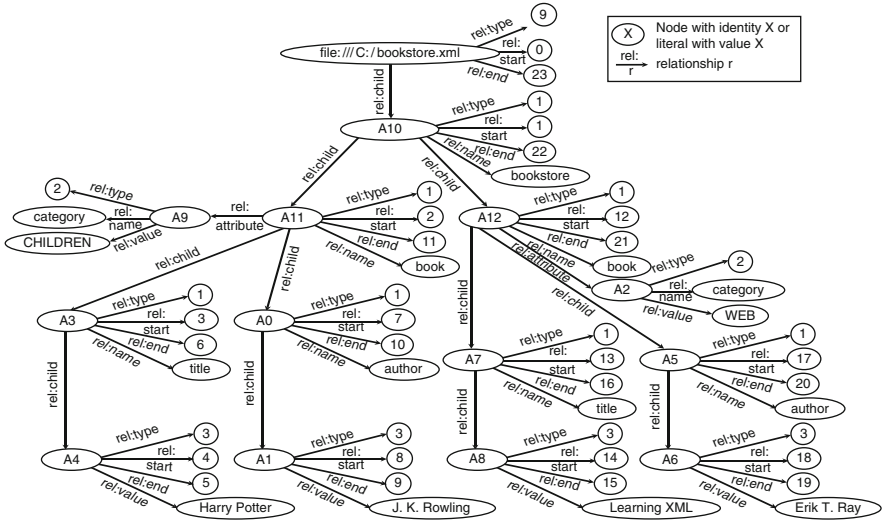


Fig. 12.25 Graphical representation of the RDF data representing the generated RDF graph of the data translation module of our prototype when using the XML data of Fig. 12.23 as input

Between any two elements a and b of the XML tree, a is a descendant node of b, if a.start>b.start and a.end<b.end. Analogously, a is an ancestor node of b, if a.start<b.start and a.end>b.end. a is a following-sibling (a preceding-sibling respectively) of b, if a.start>b.start (a.start<b.start respectively), and there exists a subject p such that p rel:child a and p rel:child b holds.

With these relationships, we can support all XPath axes in our translation scheme, as we can determine the nodes according to the basic relationships. Note that the XPath location step following::n is equivalent to ancestor-or-self::node()/following-sibling::node()/descendant-or-self::n, and the XPath location step preceding::n is equivalent to ancestor-or-self::node()/preceding-sibling::node()/descendant-or-self::n.

12.7.1.2 Translation of Queries

We translate an XPath query into an equivalent SPARQL query in the following way:

First, we determine the syntax tree of the XPath query. See Fig. 12.27, which contains the syntax tree of the XPath query of Fig. 12.26. This can be done by using standard compiler techniques, the input of which is the XPath grammar.

Second, we evaluate an attribute grammar. This attribute grammar defines computation rules for each possible situation in the syntax tree. The computation rules compute attributes of the nodes of the syntax tree. Depending on the dependencies between the attributes in the computation rules, a tree walking algorithm

/child::bookstore/parent::node()/descendant::title/child::text()

Fig. 12.26 Embedded XPath query

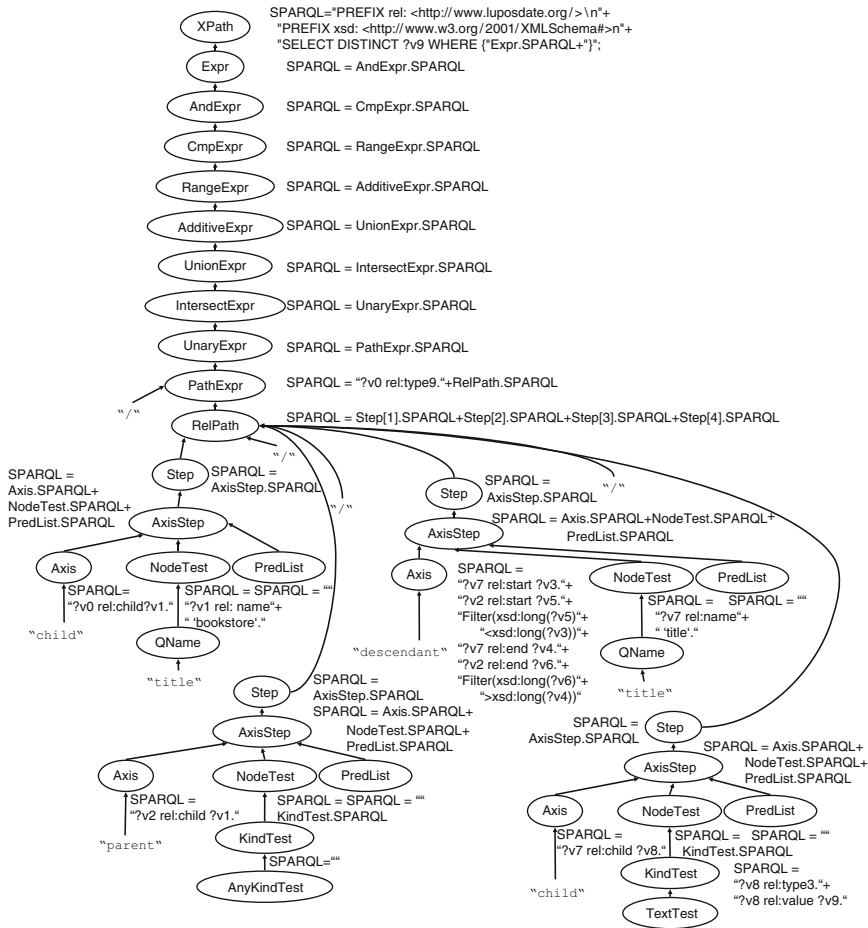


Fig. 12.27 Syntax tree of the XPath query of Fig. 12.26 and computed attributes according to our attribute grammar

defines the traversal of the syntax tree and the evaluation order of the attributes. After applying the tree walking algorithm, a special attribute SPARQL of the root node XPATH of the syntax tree contains the translated SPARQL query.

As an example of the query translation, see Fig. 12.26 for the original embedded XPath query, Fig. 12.27 for its syntax tree with computed attributes according to our attribute grammar, Fig. 12.28 for the translated SPARQL query, and Fig. 12.29 for its visual representation.

Fig. 12.28 Translated SPARQL query when using the XPath query of Fig. 12.26 as input of our prototype

```

PREFIX rel: <http://www.luposdate.org/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?v9
WHERE {
    ?v0 rel:type "9".
    ?v0 rel:child ?v1.
    ?v1 rel:name "bookstore".
    ?v2 rel:child ?v1.
    ?v7 rel:start ?v3.
    ?v2 rel:start ?v5.
    ?v7 rel:end ?v4.
    ?v2 rel:end ?v6.
    ?v7 rel:name "title".
    ?v7 rel:child ?v8.
    ?v8 rel:type "3".
    ?v8 rel:value ?v9.
    FILTER(xsd:long(?v6)>xsd:long(?v4)).
    FILTER(xsd:long(?v5)<xsd:long(?v3)).
}
    
```

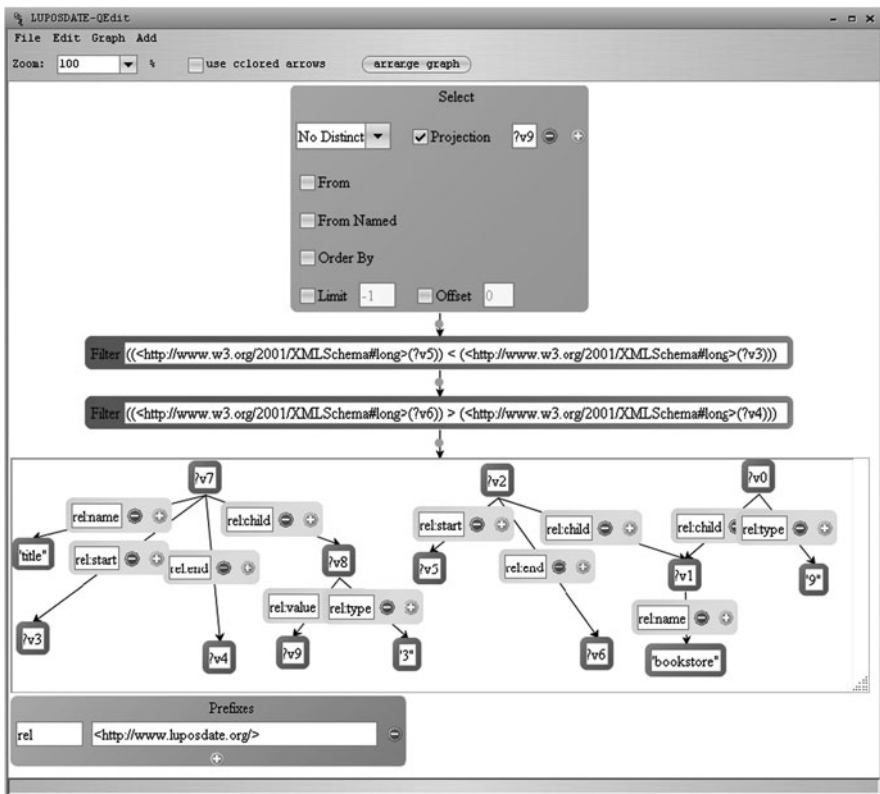


Fig. 12.29 Visual representation of the SPARQL query in Fig. 12.28

The translation of embedded XPath subqueries and the integration of their translations into the host SPARQL query require that the resultant SPARQL query containing the result of the XPath subquery is renamed according to the bound variable of the extended SPARQL expression for embedding XPath queries. Furthermore, the declared prefixes must be added to the declared prefixes of the host SPARQL query and the WHERE clause of the translated XPath query has to be added to the WHERE clause of the host SPARQL query.

12.7.1.3 Translation of Result

The result of an SPARQL query is a set of bindings of variables in the SELECT clause of an SPARQL query. We determine the translated XPath query in such a way that the retrieved bindings of the XPath query represent the resultant XML nodes of the original query. In the module of the translation of result and in the case that the result of an embedded XPath query is returned by the SPARQL query, we now rebuild the subtrees of these resultant XML nodes by considering the information of the original XML tree in the RDF data (especially the `rel:type`, `rel:child`, `rel:attribute`, `rel:namespace`, `rel:name`, `rel:value`, `rel:start`, and `rel:end` relationships). Furthermore, we sort the resultant XML trees according to the document order of the original XML tree, as the XPath language specifies the result of an XPath query to be in document order of the queried XML document. Note that the less than order relation of the computed values of `rel:start` correspond to the document order relation; that is, if a and b are the values of the `rel:start` relationship of two nodes of the translated RDF data and $a < b$, then the corresponding node of a occurs before the corresponding node of b in document order in the original XML document.

12.7.2 Performance Analysis

The test system for all experiments is a 2.66 GHz Intel Pentium 4 processor with 1 GB main memory. The test system runs Windows XP Professional Version 2002 Service Pack 2 and Java version 1.5. We have used the Java 1.5 internal XPath evaluator. Furthermore, we have used the XQuery evaluators Saxon (Kay 2010), as Saxon is widely used, and Qizx (xmlmind 2010), as Qizx is a fast evaluator, to process the original XPath queries. We have used Jena (Wilkinson et al. 2003) to process the translated SPARQL expressions, as Jena supports the current version of SPARQL and as Jena is the most widely used Semantic Web reasoning engine (see Cardoso 2007). We present the average of ten execution times of evaluating the original XPath queries, of the data translation, of the query translation, of processing the translated SPARQL queries, and of the result translation.

For the original queries, we have used the queries for the performance test of the XPathMark (Franceschet 2005) benchmark. The data are generated by the data generation tool of the benchmark, which allows scaling the size of the input data.

We have excluded those queries of the XPathMark benchmark, which are not supported by our prototype for the following reasons:

- The queries contain an XPath built-in function, which does not correspond to any built-in function of SPARQL. Our prototype supports the not, round, abs, floor, ceiling, and substring function. One possible way to support other built-in functions is to use external functions, which are especially implemented to provide the same functionality as the corresponding XPath built-in function. We did not implement these external functions, as external functions are not standardized and thus depend on the used SPARQL engine.
- The queries contain predicates of the form $[x]$, where x is a number, which restrict the current node set of a location step to its x th element. We are currently not aware of a simple, but efficient, way to access the x th element of a dynamically determined set in SPARQL queries. Note that techniques developed for SQL as described in Tatarinov et al. (2002) cannot be adapted to SPARQL, as a RANK clause as in SQL or an equivalent language construct is missing in the SPARQL language.

Figure 12.30 presents the execution times of the original XPath queries of the XPathMark benchmark, of the data translation, of the query translation, of the translated SPARQL query, and of the result translation when using an input XML document of size 56.55 kB. Furthermore, Figure 12.30 presents the average execution times of all these 18 queries of the XPathMark benchmark (most right column). Figure 12.31 present the average execution times when varying the size of the input file. Furthermore, it shows that the average execution times for processing the translated SPARQL queries are dominated by the execution time of those translated queries, which are translated from XPath queries containing a recursive axis such as descendant, descendant-or-self, ancestor, ancestor-or-self, following, preceding, following-sibling, and preceding-sibling. The translated SPARQL queries of XPath queries containing a recursive axes have filter expressions such as FILTER (xsd:long(?v6) > xsd:long(?v4)), the processing of which is time-consuming. Future versions of Jena or other SPARQL engines may optimize these kinds of filter expressions, such that the translations for recursive axes are faster processed.

12.8 Related Work

There are many contributions (e.g. Florescu and Kossmann 1999; Georgiadis and Vassalos 2006; Manolescu et al. 2001; Shanmugasundaram et al. 1999; Subramanyam, and Kumar 2005; Fan et al. 2005; Yoshikawa et al. 2001; Grust et al. 2004; Tatarinov et al. 2002) to source-to-source translations for evaluating XPath expressions on relational database management systems. Many techniques described there can be adapted for evaluating XPath expressions on SPARQL evaluators such as using a numbering scheme for the XML
(continued)

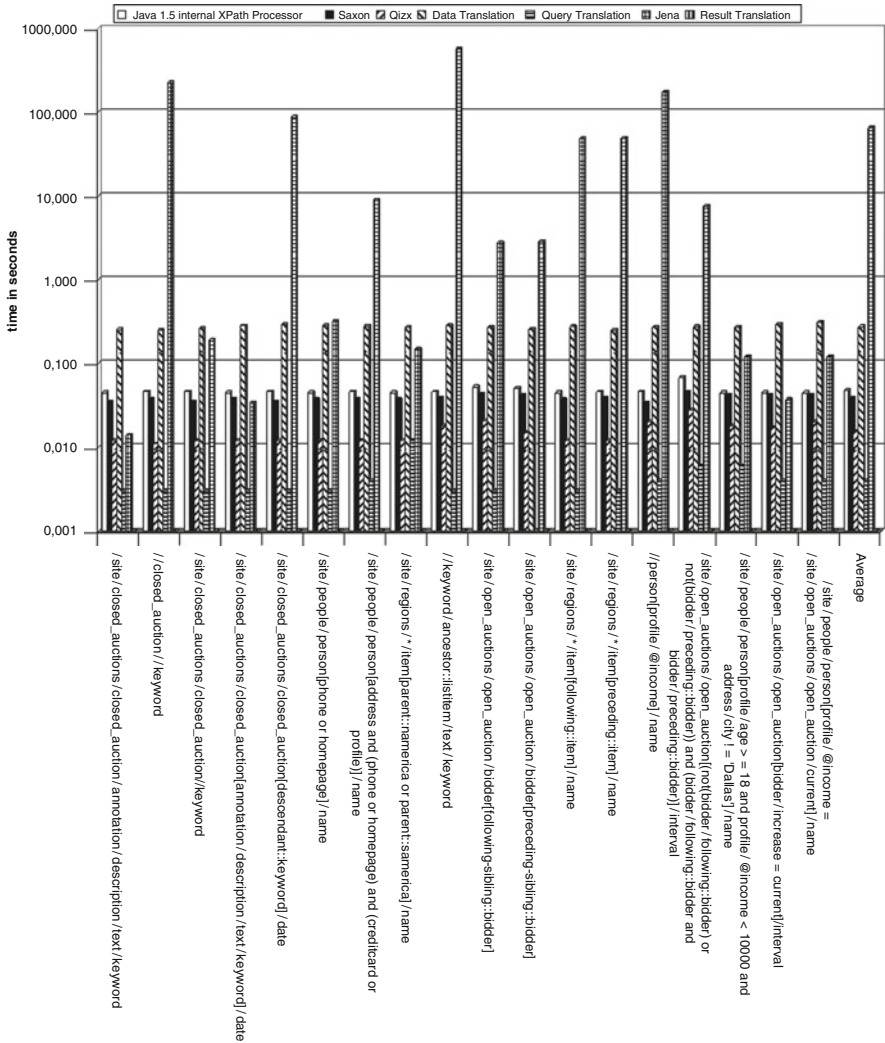


Fig. 12.30 Execution time of the original XPath queries of the XPathMark Benchmark, of the data translation, of the query translation, of the translated SPARQL query, and of the result translation when using an input XML document of size 56.55 kB

data to support all XPath axes (Grust et al. 2004), but some other techniques cannot be adapted such as the evaluation of positional predicates (Tatarinov et al. 2002) as SQL supports more language constructs than SPARQL such as the rank clause.

Bettentrupp et al. (2006), Fokoue et al. (2005), Klein et al. (2005), Lechner et al. (2001), Groppe et al. (2011c), and Groppe et al. (2009c) focus on the

(continued)

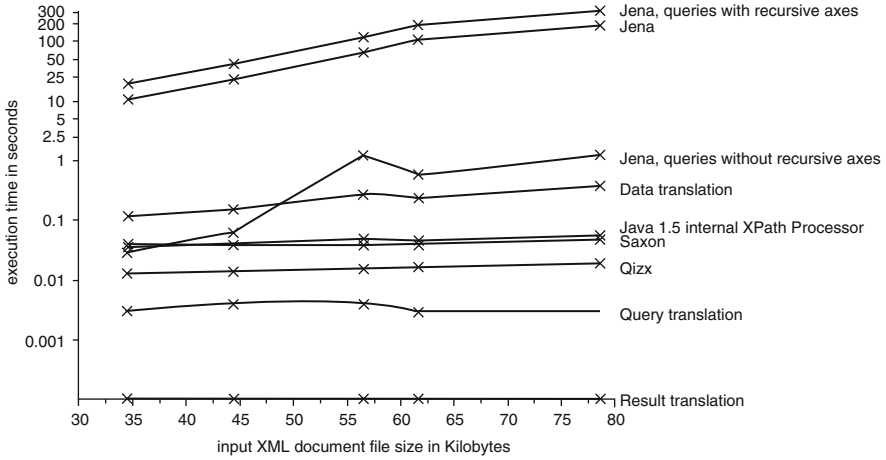


Fig. 12.31 Average execution times of the original queries of the XPathMark benchmark, of data translation, query translation, result translation, and the execution times of the translated SPARQL queries (all, only those containing recursive axes, and those which contain only nonrecursive queries) using Jena

translations between XSLT and XQuery (and vice versa), which embed the XPath language.

Furthermore, some contributions deal with bridging the gap between SPARQL/RDF and the relational world (e.g., Chong et al. 2005; Dokulil 2006; Harris and Shadbolt 2005; Laborda and Conrad 2006).

Polleres (2007) describes a translation scheme from SPARQL to rules by developing transformation rules from a SPARQL algebra to rules.

12.9 Summary and Conclusions

XQuery and SPARQL are query languages developed for and by different communities: the XML community in the case of XQuery and the Semantic Web community in the case of SPARQL. Nevertheless, translations between their data models XML and RDF and embedding the query language SPARQL into XQuery and XSLT and embedding XPath queries into SPARQL queries are possible. Translations between RDF and XML data and embedding SPARQL into XQuery/XSLT and XPath into SPARQL enable XML tools to deal with RDF data and to process SPARQL queries, and SPARQL query evaluators to deal with XML data and to process XPath queries as subqueries.

We have developed prototypes for translating SPARQL queries into XQuery queries and into XSLT stylesheets as well as translating XPath queries into SPARQL queries. The prototypes show that a wide range of queries can be translated and performed efficiently.

Chapter 13

Summary, Conclusions, and Future Work

This book covers a wide range of topics in the area of the Semantic Web related to query processing.

First of all in Chap. 2 after the introduction in Chap. 1, we have learnt about the basic specifications of the Semantic Web, its data language RDF, its ontology languages RDF Schema and OWL, its query language SPARQL, and its rule language RIF. We got to know that the various different Semantic Web languages are powerful enough for nearly any kind of application.

Then the B⁺-tree has been introduced as an efficient index for large-scale datasets (Chap. 3). Also, external sorting for efficiently building the B⁺-tree from large-scale datasets from scratch has been discussed. These are the basics for efficient indexing and querying large-scale datasets, which are the topics in later chapters.

An overview of query optimization phases has been given afterward (Chap. 4). We have then provided the transformation from SPARQL to a core of SPARQL without redundancies in the language constructs as basic operation used in many tools to simplify, for example, building an operatorgraph or a visual query.

An algebra for SPARQL and its logical optimizations are discussed in Chap. 5. Applying equivalency rules to the query expressed in the notion of the algebra is the basis for any kind of optimization.

The algorithms for processing of the operators of a SPARQL query and indexing techniques for large-scale as well as small datasets are the content of Chap. 6. Experimental results show the superior performance for large-scale as well as small datasets.

The processing of infinite data streams as generated, for example, from sensors is the topic of Chap. 7. Infinite data streams require another kind of operators that periodically calculate query results. Because differences to previous query results are calculated internally, the traditional operators for joins, sorting, and so on must also deal with requests to delete a solution from their temporary indices and buffers. The demonstrated solution shows the practical relevance of this young field of research.

Chapter 8 introduces into the world of multicore processors and their optimizations for Semantic Web parallel databases. Not all queries can benefit from parallel

processing. However, the query optimizer can already estimate if an operator is worth to be parallelized and can generate the operator graph accordingly.

Inference is a highly costly operation, which needs optimization. Inference materialization strategies as well as optimization approaches have been discussed in Chap. 9. The optimizations work for database queries as well as for stream queries and significantly improve the performance.

Visual query languages to help users formulating queries are introduced in Chap. 10. Users do not need to learn the syntax of SPARQL and the visual representation helps them to easily see connected terms in their query. Furthermore, the system can provide suggestions to extend and refine their queries.

Chapter 11 discusses the features, possibilities, and the technology behind embedded Semantic Web languages in programming languages. The advantage is that a static analysis can detect already many errors during compilation, which may be otherwise only detected after extensive tests during runtime. By determining the types of query results already at compile time, the java type system can also be orthogonally used to detect this kind of type errors. Therefore, embedded query languages ensure more stable programs and applications.

Chapter 12 compares the XML world as alternative to the Semantic Web world and shows common features as well as differences. Furthermore, this chapter shows that it is possible to embed SPARQL into XQuery and into XSLT as well as to embed XPath into SPARQL. We can conclude that you can theoretically use XML or the Semantic Web (or both in parallel) for a solution involving data and queries. However, some solutions are more elegant and easier to formulate when using XML with its tree model or the Semantic Web with its graph model.

This book already covers many aspects of data management and query processing. It demonstrates that the technologies for a high performance Semantic Web have been developed. These (or even more advanced) technologies must now find their ways into industry and their commercial products. Only then the Semantic Web will be successful and accepted by the user and at the market. Many companies such as Oracle have already started to support Semantic Web technologies, but it is still a long way until it is a *must* for companies to support Semantic Web technologies.

13.1 Possibilities for Future Work

A book can cover only a snapshot of current research. In the following paragraphs, we will give hints where future work can be done on the presented topics.

Of course, future work can deal with more equivalency rules for logical optimizations in the operator graph as well as for inference and specialized equivalency rules for stream queries.

Stream processing can be further extended by developing new types of windows and update streams with the possibility to delete or update an older triple.

Parallel algorithms can be further investigated for remaining operators. However, the most rewarding operators to be parallelized are the join operators, which have already been discussed in this book.

The obvious next step for research on inference is to support OWL and OWL 2, where probably further optimization rules must be developed.

Research on visual query languages can include research on further simplifying query creation by having a more browser-like graphical user interface without losing the provided flexibility by, for example, using the current approach as fallback to further manipulate the visual query.

Research on embedded languages can focus on using the static analysis to optimize queries and the program containing the queries together instead separately, which is state of the art. This promises much better optimization possibilities. Furthermore, also the costly inference could be precomputed as much as possible based on a static analysis and optimized together with the program code.

The relationship to XML can be further investigated and, for example, inference can be included to be simulated by XQuery and/or XSLT. Other data models such as the object-oriented one and query languages such as object-oriented query languages can be investigated if they are as powerful as the Semantic Web ones and how to transform data and translate queries into each other.

Another big topic would be to efficiently support rules (e.g., RIF) and focus on optimizations for rules in large-scale databases. It is also an open research question how to combine inference based on ontologies as well as based on rules and query processing and apply all these three different kinds of processing in one system. Research must be further driven to rule stream processing, parallel rule processing, visual rules, and embedded rules in programming languages.

Microformats are used to embed RDF data into webpages. Research is missing which can deal with these highly distributed and split pieces of data, their retrieval, integration, and processing.

A new research trend in databases is to use hardware such as FPGAs or the power of graphical processors for optimizing processing. New optimization techniques have to be developed to optimize Semantic Web queries, inference, and rules on this hardware, but the benefits seem to be enormous.

References

- Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the Borealis stream processing engine. In: Proceedings International Conference on Innovative Data Systems Research (CIDR 2005), 2005
- Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning, VLDB, Vienna, Austria (2007)
- Acciari, A., Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Palmieri, M., Rosati, R.: QuOnto: querying ontologies. In: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005), Pittsburgh, USA (2005)
- Adida, B., Birbeck, M. (eds): RDFa primer – bridging the human and data webs, W3C Working Group Note. <http://www.w3.org/TR/xhtml-rdfa-primer/>, 14 October 2008
- Alex Sung L.G., Ahmed N., Blanco R., Li H., Soliman M.A., Hadaller D.: A survey of data management in peer-to-peer systems. Web Data Management (2005)
- Alvestrand, H.: RFC 3066 - Tags for the identification of languages, IETF, <http://www.isi.edu/in-notes/rfc3066.txt> (2001)
- Amdahl, G.: Validity of the single processor approach to achieving large-scale computing capabilities. AFIPS Conference Proceedings (30), pp. 483–485 (1967)
- Angles, R., Gutiérrez C.: Querying RDF data from a graph database perspective. In: ESWC (2005)
- Angles, R., Gutiérrez C.: The expressive power of SPARQL, In: 7th International Semantic Web Conference (ISWC 2008), Karlsruhe, Germany (2008)
- ANSI, Information technology – Database languages – SQL – Part 0:SQL/OLB standard. ANSI X3.135 (1998)
- Apache Software Foundation, Cocoon, <http://cocoon.apache.org> (2009)
- Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J. STREAM: The Stanford Stream Data Manager (Demonstration Description). In: Proceedings of ACM International Conference on Management of Data (SIGMOD 2003), p. 665 (2003)
- Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB Journal **15**(2), 121–142 (2006)
- Auer, S., et al.: Dbpedia: a nucleus for a web of open data. In: ISWC/ASWC (2007)
- Avgustinov, P., Hajiyeve, E., Ongkingco, N., Demoor, O., Sereni, D., Tibble, J., Verbaere, M.: Semantics of static pointcuts in AspectJ. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07). ACM Press, New York, pp. 11–23 (2007)
- Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): Description logic handbook: theory, implementation, and applications, 2nd edn. Cambridge, The University Press, Cambridge (2007)
- Babu, S., Widom, J.: Continuous queries over data streams. SIGMOD Rec **30**(3), 109–120 (2001)

- Bai, Y., Thakkar, H., Wang, H., Luo, C., Zaniolo, C.: A data stream language and system designed for power and extensibility. In: Proceedings International Conference on Information and Knowledge Management (CIKM 2006), pp. 337–346 (2006)
- Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., Zdonik, S.: Retrospective on Aurora. *The VLDB J* **13**(4), 370–383 (2004)
- Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs (extended abstract). In: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of Database Systems. Cambridge, Massachusetts (1986)
- Bancilhon, F., Ramakrishnan, R.: An amateur’s introduction to recursive query processing of strategies. In: Proceedings of the ACM-SIGMOD International Conference on Management Data. ACM, New York (1986)
- Barbieri, D.F., Braga D., Ceri S., Della Valle, E., Grossniklaus M., C-SPARQL: SPARQL for continuous querying. In: Proceedings of the 18th International Conference on World Wide Web (WWW 2009), Madrid, Spain (2009)
- Barbieri D.F., Braga D., Ceri S., Grossniklaus M.: An execution environment for C-SPARQL queries, In: 13th International Conference on Extending Database Technology (EDBT 2010), Lausanne, Switzerland (2010)
- Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. *Acta Informatica* **1**, 173–189 (1972)
- Bechhofer, S., Horrocks, I., Turi, D.: The OWL instance store: system description. In: Proceedings of the 20th International Conference on Automated Deduction (CADE 2005), Tallinn, Estonia (2005)
- Beckett, D. (ed): RDF/XML syntax specification (Revised), W3C Recommendation, 10th February 2004
- Beckett, D.: The design and implementation of the Redland RDF application framework. In: WWW (2001)
- Beckett, D.: Turtle – terse RDF triple language. <http://www.dajobe.org/2004/01/turtle/> (2006)
- Beckett, D., Broekstra, J., (eds): SPARQL query results XML format, W3C Recommendation. 15 January 2008, <http://www.w3.org/TR/rdf-sparql-XMLres/> (2008)
- Beeri, C., Ramakrishnan, R.: On the power of magic. In: Proceedings of the ACM SIGACTSIGMOD Symposium on Principles of Database Systems. ACM, New York, pp. 269–283 (1987)
- Bege-Dove, G., Brickley, D., Dornfest, R., Davis, I., Dodds, L., Eisenzopf, J., Galbraith, D., Guha, R.V., MacLeod, K., Miller, E., Swartz, A., van der Vlist, E.: RDF site summary (RSS) 1.0, <http://purl.org/rss/1.0/spec> (2001)
- Berners-Lee, T.: N3QL – RDF data query language, W3C, July 2004. <http://www.w3.org/DesignIssues/N3QL.html> (2004)
- Berners-Lee, T.: Notation 3 – An RDF language for the semantic web. W3C. <http://www.w3.org/DesignIssues/Notation3.html> (1998)
- Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web, *Scientific American Magazine*, May 2001
- Berners-Lee, T., Fielding, R., Masinter, L.: Uniform resource identifiers (URI): Generic syntax, RFC 2396 (1998)
- Bernstein, A., Stocker, M., Kiefer, C.: SPARQL query optimization using selectivity estimation. ISWC (2007)
- Bettentrupp, R., Groppe, S., Groppe, J., Böttcher, S., Gruenwald, L.: A prototype for translating XSLT into XQuery, In: Eighth International Conference on Enterprise Information Systems (ICEIS 2006), Paphos, Cyprus (2006)
- Boley, H.: RIF RuleML Rosetta Ring: Round-Tripping the Dlex Subset of Datalog RuleML and RIF-Core, Rule Interchange and Applications, International Symposium, RuleML 2009. Las Vegas, Nevada, USA, November 5–7 (2009)
- Boley, H., Kifer M., (eds): RIF Basic Logic Dialect, W3C candidate recommendation, 1 October 2009. <http://www.w3.org/TR/rif-bl/> (2009)

- Boley, H., Hallmark, G., Kifer, M., Paschke A., Polleres A., Reynolds D., (eds.): RIF Core Dialect, W3C Candidate Recommendation, 1 October 2009. <http://www.w3.org/TR/rif-core/> (2009)
- Bolles, A., Grawunder, M., Jacobi, J.: Streaming SPARQL – Extending SPARQL to process data streams. In: Proceedings of Europe Semantic Web Conference (ESWC 2008), Tenerife, Canary Islands, Spain (2008)
- Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., Valduriez, P.: Prototyping Budda: a highly parallel database system. *IEEE Knowledge and Data Engineering*, vol. 2, no. 1, March (1990)
- Borsje, J., Embregts H.: Graphical query composition and natural language processing in an RDF visualization interface. In: Erasmus School of Economics and Business Economics, vol. Bachelor, Rotterdam (2006)
- Bost, T., Bonnard, P.: Mark proctor: implementation of production rules for a RIF dialect: A MISMO proof-of-concept for loan rates. International Symposium on Advances in Rule Interchange and Applications (RuleML), Orlando, Florida (2007)
- Brabrand, C., Møller, A., Schwartzbach, M.I.: The <bigwig> project. *ACM ToIT* **2**, 79–114 (2002)
- Brickley, D., Guha, D.V.: RDF vocabulary description language 1.0: RDF Schema, W3C Recommendation, <http://www.w3.org/TR/rdf-schema/> (2004)
- Brickley, D., Miller L.: FOAF vocabulary specification 0.9, <http://xmlns.com/foaf/spec> (2007)
- Broekstra, J., Kampman A.: SeRQL: A second generation RDF Query language. User manual, Aduna, 2003. <http://sesame.aduna.biz/publications/SeRQLmanual.html> (2003)
- Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF schema. In: International Semantic Web Conference 2002, Chia, Sardinai, Italy (2002)
- Buneman, P., Ohori, A.: Polymorphism and type inference in database programming. *ACM ToDS* **21**(1), 30–76 (1996)
- Bussche, J., Waller, E.: Type inference in the polymorphic relational algebra. PODS, Philadelphia, USA (1999)
- Cail M., Frank M.: RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network, WWW 2004, Manhattan, NY, USA (2004)
- Cardoso, J.: The Semantic Web Vision: Where are We?, *IEEE Intelligent Systems*, pp. 22–26 (2007)
- Catarci, T., Dongilli, P., Di Mascio, T., Franconi, E., Santucci, G., Tessaris, S.: An ontology based visual tool for query formulation support, ECAI 2004, Valencia (2004)
- Chaudhuri, S.: An overview of query optimization in relational systems. In: Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS98), Seattle, Washington, USA (1998)
- Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. *J ACM* **43**(1), 20–74 (1996)
- Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A scalable continuous query system for internet databases. In: Proceedings of ACM International Conference on Management of Data (SIGMOD 2000), pp. 379–390 (2000)
- Chen, C., Haarslev, V., Wang, J.: LAS: extending racer by a large Abox store. In: Proceedings of the International Workshop on Description Logics (DL'05), Edinburgh, Scotland, UK (2005)
- Chinnici, R., Moreau, J.-J., Ryman, A., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Recommendation. <http://www.w3.org/TR/wsdl20/>, 26 June 2007
- Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient SQL-based RDF querying scheme, VLDB (2005)
- Christensen, A.S., Møller, A., Schwartzbach, M.I.: Extending Java for high-level web service construction. *ACM ToPLaS*, **25**(6) (2003)
- Clark K.G., Feigenbaum L., Torres E. (ed): SPARQL protocol for RDF, W3C Recommendation, 15 January 2008. <http://www.w3.org/TR/rdf-sparql-protocol/> (2008)

- Cole, R.: Parallel merge sort. *SIAM J Comput* **17**(4), 770–785 (1998)
- Connolly, T., Begg, C.: *Database Systems – A practical approach to design, implementation and management*, 3rd edition, Addison Wesley (2002)
- Cook, S.A.: *The complexity of theorem-proving procedures*. ACM STOC, Shaker Heights, Ohio, United States (1971)
- Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to algorithms*, MIT Press (1990)
- Cyganik, R.: *A relational algebra for sparql*. Technical report HPL-2005-170, HP Laboratories Bristol (2005)
- Czejdo, B., Elmasri, R., Rusinkiewicz, M., Embley, D.: An algebraic language for graphical query formulation using an EER model. *ACM Conference on Computer Science* (1987)
- Das, S.K., Wen-Bing, H., Moon, G.S.: An efficient algorithm for managing a parallel heap. *International Journal of Parallel, Emergent and Distributed Systems* **4**(3), 281–299 (1994)
- de Bruijn, J.: RIF RDF and OWL Compatibility, W3C Candidate Recommendation 1 October 2009. http://www.w3.org/TR/rif-rdf-owl/#Importing_RDF_and_OWL_in_RIF (2009)
- de Kunder, M.: The size of the World Wide Web, <http://www.worldwidewebsite.com/> (2010)
- de Laborda, C.P., Conrad, S.: *Bringing relational data into the SemanticWeb using SPARQL and Relational.OWL*. SWDB'06, Atlanta, Georgia, USA (2006)
- de Sainte Marie C.: A modest proposal to enable RIF dialects with limited forward compatibility. Rule interchange and applications. In: *International Symposium, RuleML 2009*, Las Vegas, Nevada, USA, November 5–7 (2009)
- de Sainte Marie, C.; Paschke, A., Hallmark, G., (eds): RIF production rule dialect. W3C candidate recommendation, 1 October 2009. <http://www.w3.org/TR/rif-prd/>
- de Sainte Marie, C.: W3C rule interchange format – The production rule dialect, tutorial at RuleML, slides available at <http://intranet.cs.man.ac.uk/ruleML/presentations/standards2.ppt> (2008)
- De Troyer, O., Meersman, R., Verlinden, P.: RIDL on the CRIS case: a workbench for NIAM. IFIP.8.1. (1988)
- Dean, M., Schreiber, G. (Eds): *OWL Web Ontology Language Reference*, W3C Recommendation, 10 February 2004, <http://www.w3.org/TR/2004/REC-owl-ref-20040210/> (2004)
- DeHaan, D., Tompa, F.W.: Optimal top-down join enumeration. *SIGMOD*, Beijing, China (2007)
- Deo, N., Prasad, S.: Parallel heap: an optimal parallel priority queue. *J Supercomput* **6**(1), 1992 (1992)
- DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. *Communications of ACM* **35**(6), 85–98 (1992)
- DeWitt, D.J., Gerber, R.H., Graefe, G., Heytens, M.L., Kumar, K.B.: *GAMMA – A high performance dataflow database machine*. VLDB, 1986
- Dokulil, J.: Evaluation of SPARQL queries using relational databases. *ISWC*, Athens, GA, USA (2006)
- Droop, M., Flarer, M., Groppe, J., Groppe, S., Linnemann, V., Pinggera, J., Santner, F., Schier, M., Schöpf, F., Staffler, H., Zugal, S.: Translating XPath Queries into SPARQL Queries, On the Move (OTM 2007) Federated Conferences and Workshops (DOA, ODBASE, CoopIS, GADA, IS). In: *6th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2007)*, Vilamoura, Algarve, Portugal (2007)
- Droop, M., Flarer, M., Groppe, J., Groppe, S., Linnemann, V., Pinggera, J., Santner, F., Schier, M., Schöpf, F., Staffler, H., Zugal, S.: Bringing the XML and Semantic Web Worlds Closer: Transforming XML into RDF and Embedding XPath into SPARQL. In: Filipe, J., Cordeiro, J. (Eds), *Enterprise Information Systems*, 10th International Conference, ICEIS 2008, Barcelona, Spain, June 12–16, 2008, Revised Selected Papers, *Lecture Notes in Business Information Processing*, Springer, Heidelberg (2009)
- Droop, M., Flarer, M., Groppe, J., Groppe, S., Linnemann, V., Pinggera, J., Santner, F., Schier, M., Schöpf, F., Staffler, H., Zugal, S.: Embedding XPath Queries Into SPARQL Queries, 10th International Conference on Enterprise Information Systems (ICEIS 2008), Barcelona, Spain (2008)

- Dürst, M., Suignard, M.: Internationalized resource identifiers (IRIs), <http://www.ietf.org/rfc/rfc3987.txt>, W3C Memo (2005)
- eBay, eBay Developers Program, <http://developer.ebay.com/> (2010)
- eBay, eBay, <http://www.ebay.com/> (2010)
- Edutella, Edutella – P2P for the Semantic Web, <http://www.edutella.org/edutella.shtml> (2004)
- Eiter, T., Ianni, G., Krennwallner, T., Polleres, A.: Rules and ontologies for the Semantic Web, Reasoning Web 2008, LNCS 5224, pp. 1–53 (2008)
- Elmasri, R., Navathe, S.B.: Fundamentals of database systems, 3rd edn, Addison Wesley (2000)
- Erdmann, M. (ed): GNADE User’s Guide: GNADE, The GNat Ada Database Environment; Version 1.5.3. (2002)
- Faber, W., Greco, G., Leone, N.: Magic sets and their application to data integration. In: 10th International Conference of Database Theory (ICDT). Edinburgh, UK (2005)
- Fadhil A., Haarslev V.: GLOO: a graphical query language for OWL ontologies, OWL: Experience and Directions, pp. 215–260. (2006)
- Fan, W., Yu, J.X., Lu, H., Lu, J., Rastogi, R.: Query translation from XPath to SQL in the presence of recursive DTDs, *VLDB*, Trondheim, Norway (2005)
- Feigenbaum, L., (ed): DAWG Testcases, <http://www.w3.org/2001/sw/DataAccess/tests/r2>, 2008.
- Fell, K., Kalis, F., Samsel M.: LUPOS eBay-Stream-Reader 2010, <http://lupos.metawort.de/> (2010)
- Fernández, M., Robie, J. (Eds): XQuery 1.0 and XPath 2.0 Data Model, W3C Working Draft, <http://www.w3c.org/TR/2001/WD-query-datamodel> (2001)
- Fielding, R.T.: Architectural styles and the design of network-based software architectures, PhD Thesis, University of California, Irvine (2000)
- Florescu, D., Kossmann, D.: Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin* 22 (1999) 27–34 (1999)
- Fokoue, A., Rose, K., Siméon, J., Villard, L.: Compiling XSLT 2.0 into XQuery 1.0, WWW 2005, Chiba, Japan (2005)
- Fokoue, A., Kershenbaum, A., Ma, L., Schonberg, E., Srinivas, K.: The Summary Abox: Cutting Ontologies Down to Size. In: Proceedings of the 5th International Semantic Web Conference (ISWC’06), Athens, GA, USA (2006)
- Franceschet, M.: XPathMark – An XPath Benchmark for the XMark Generated Data. *XSym* 2005, Trondheim, Norway (2005)
- Frasincar, F., Houben, G.J., Vdovjak, R., Barna, P.: RAL: an Algebra for Querying RDF. WWW, New York, USA (2004)
- Friend, E.H.: Sorting on electronic computer systems. *J ACM* 3(3) (1956)
- Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall, Upper Saddle River, NJ (2002)
- Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming: Proceedings of the Fifth Conference and Symposium. pp. 1070–1080. (1988)
- Georgiadis, H., Vassalos, V.: Improving the efficiency of XPath execution on relational systems. *EDBT*, vol. 3896, pp. 570–587, Springer, Berlin (2006)
- Gilmore, W., Black, C., Clegg, D., Dayal, S., Fourt, E., Goodman, S., Richey, J., Smith, G., Swift P.: Open Client Embedded SQL/COBOL Programmer’s Guide. SYBASE Embedded SQL Release 10.0. (1994)
- Golab L., Johnson T., Koudas N., Srivastava D., Toman D.: Optimizing away joins on data streams. In: Proceedings of International Workshop on Scalable Stream Processing System (SSPS 2008), pp. 48–57. (2008)
- Gordon T.F.: Guido Governatori, and Antonino Rotolo, Rules and Norms: Requirements for rule interchange languages in the legal domain. Rule Interchange and Applications, International Symposium, RuleML 2009. Las Vegas, Nevada, USA, November 5–7 (2009)
- Graefe, G.: Query evaluation techniques for large database. *ACM Computing Surveys* 25, 2 (June), 73–170 (1993)

- Grant, J., Beckett, D. (eds): RDF Test Cases, W3C Recommendation, <http://www.w3.org/TR/rdf-testcases/>, 10th February 2004
- Griffiths-Selinger, P., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database system. ACM SIGMOD (1979)
- Groppe S.: Mobile and distributed databases. Lecture. <http://www.ifis.uni-luebeck.de/index.php?id=mobile-ws0910>. (2010)
- Groppe, S., Groppe, J.: LUPOSDATE demonstration, <http://www.ifis.uni-luebeck.de/index.php?id=luposdate-demo> (2009)
- Groppe, S.: Mobile and distributed databases. Lecture. <http://www.ifis.uni-luebeck.de/index.php?id=mobile-ws0809> (2009)
- Groppe, S., Groppe, J., Linnemann, V.: Using an index of precomputed joins in order to speed up SPARQL processing. In: 9th International Conference on Enterprise Information Systems (ICEIS 2007), Funchal, Portugal (2007a)
- Groppe, S., Groppe, J., Kukulenz, D. Linnemann, V.: A SPARQL engine for streaming RDF data. In: 3rd International Conference on Signal-Image Technology and Internet-Based Systems (SITIS 2007). Shanghai, China, (2007b). This paper received an honorable mention at the SITIS'07 Conference 2007
- Groppe, S., Neumann, J.: Demonstration of SWOBE. http://www.ifis.uni-luebeck.de/~groppe/swobe_demo/ (2008)
- Groppe, S., Groppe, J., Linnemann, V., Kukulenz, D., Hoeller, N., Reinke, C.: Embedding SPARQL into XQuery/XSLT. In: 23rd ACM symposium on applied computing (ACM SAC 2008), Fortaleza, Ceara, Brasilien (2008)
- Groppe, J., Groppe, S., Ebers, S., Linnemann, V.: Efficient Processing of SPARQL joins in memory by dynamically restricting triple patterns, *ACM SAC*, Waikiki Beach, Honolulu, Hawaii, USA (2009)
- Groppe, J., Groppe, S., Schleifer, A., Linnemann, V.: LuposDate: A semantic web database system. In: 18th ACM conference on information and knowledge management (ACM CIKM 2009), Hong Kong, China (2009)
- Groppe, S, Groppe, J, Reinke, C., Hoeller, N., Linnemann, V.: XSLT: Common issues with XQuery and special issues of XSLT. In: Eric Pardede (Ed.), Open and novel issues in XML database applications: future directions and advanced technologies, IGI Global (2009)
- Groppe, J., Groppe, S., Kolbaum, J.: Optimization of SPARQL by Using coreSPARQL, In: Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS 2009). Milano, Italy (2009d). 2009
- Groppe, S, Neumann, J, Linnemann, V.: SWOBE – Embedding the Semantic Web languages RDF, SPARQL and SPARUL into Java for Guaranteeing Type Safety, for Checking the Satisfiability of Queries and for the Determination of Query Result Types, 24th ACM Symposium on Applied Computing (ACM SAC 2009), Waikiki Beach, Honolulu, Hawaii, USA (2009e)
- Groppe, S, Groppe, J.: External sorting for index construction of large semantic web databases. In: 25th Symposium On Applied Computing (ACM SAC 2010), Sierre, Switzerland (2010)
- Groppe, J., Groppe, S.: Parallelizing join computations of SPARQL queries for large semantic web databases. In: 26th symposium on applied computing (ACM SAC 2011), TaiChung, Taiwan (2011a)
- Groppe, J., Groppe, S., Schleifer, A.: Visual query system for analyzing social semantic Web. In: 20th International World Wide Web Conference (WWW 2011), Hyderabad, India (2011b)
- Groppe, S., Groppe, J., Klein, N., Bettentrupp, R., Böttcher, S., Gruenwald, L.: Transforming XSLT stylesheets into XQuery expressions and vice versa. *Computer Languages, Systems and Structures Journal* **37**(3), 76–111 (2011c)
- Grust, T., van Keulen, M., Teubner, J.: Accelerating XPath evaluation in any RDBMS. *ACM Trans Database Syst* **29**, 91–131 (2004)
- Guha, R.V.: rdfDB : An RDF database. <http://www.guha.com/rdfdb/> (2010)

- Guha, R., McCool, R.: TAP: A semantic web test-bed. *Journal of Web Semantics*, vol. 1, Issue 1, December (2003)
- Guha, R.V., McCool, R., Fikes, R.: Contexts for the Semantic Web. In: *Proceedings of the 3rd International Semantic Web Conference*, Hiroshima, Japan, November (2004)
- Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. *Web Semantics* 3(2) (2005)
- Gutierrez, C., Hurtado, C., Mendelzon, A.: *Foundations of Semantic Web Databases*. PODS 2004, Paris, France (2004)
- Haarslev, V., Möller, R.: Racer: A core inference engine for the Semantic Web Ontology Language (OWL). In: *Proceeding of the 2nd International Workshop on Evaluation of Ontology-based Tools*. (2003)
- Hallmark, G., de Sainte Marie, C., Del Fabro, M.D., Albert, P., Paschke, A.: Please pass the rules: A rule interchange demonstration. In: *International Symposium on Rule Interchange and Applications (RuleML)*. Orlando, FL, USA (2008)
- Harris, S., Gibbins, N.: 3store: Efficient bulk RDF storage. In: *PSSS*. (2003)
- Harris, S., Shadbolt, N.: SPARQL query processing with conventional relational database systems. *WISE Workshops 2005*, New York, USA (2005)
- Harth, A., Decker, S.: Optimized index structure for querying RDF from the web. In: *Proceedings of the 3rd Latin American Web Congress (LA-WEB)*, Buenos Aires, Argentina (2005)
- Harth, A., Kruk, S., Decker, S.: Graphical representation of RDF queries, *World Wide Web*. (2006)
- Hawke, S.: RIF: bringing order to chaos, keynote at RuleML, slides. [http://www.w3.org/2009/Talks/1105-ruleml/#\(1\)](http://www.w3.org/2009/Talks/1105-ruleml/#(1)) (2009)
- Hayes, J., Gutiérrez C.: Bipartite graphs as intermediate model for RDF. In: *ISWC*, (2004)
- Hayes, P.: RDF Semantics, W3C Recommendation, 10th February 2004, <http://www.w3.org/TR/rdf-mt/>
- Heese, R.: Query graph model for SPARQL, *ER Workshops 2006 Tucson, AZ, USA* (2006)
- Henschen, L.J., Naqvi, S.A.: Compiling queries in relational first-order databases. *J ACM* 31(1), 47–85 (1984)
- Hoare, C.A.R.: Monitors: an operating system structuring concept. *Commun. ACM* 17(10), 549–557 (1974)
- Hofstede, A., Proper, H., van der Weide, T.: Computer supported query formulation in an evolving context. In: *Proceedings of the ADC*, (1995)
- Hogenboom, F., Milea, V., Frasinca, F., Kaymak, U.: RDF-GL: A SPARQL-Based Graphical Query Language for RDF. In: *Emergent Web Intelligence: Advanced Information Retrieval*. Springer (2010)
- Hosoya, H., Frisch, A., Castagna, G.: Parametric polymorphism for XML. In: *POPL*, Long Beach, USA (2005)
- IBM, IBM RIF Demo, *International Symposium on Rule Interchange and Applications (RuleML)*, Orlando, FL, USA (2008)
- IBM. *IBM Informix ESQL/C Programmer's Manual*. Version 9.53, IBM, (2003)
- Ingres Corporation. *Ingres® 2006 Embedded SQL Companion Guide*. (2006)
- International Organization for Standardization (ISO), ISO/IEC 10646:2003: Information technology – Universal Multiple-Octet Coded Character Set (UCS), <http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=39921&ICS1=35&ICS2=40&ICS3> (2003)
- International Organization for Standardization (ISO), ISO/IEC 14977:1996: Information technology – Syntactic metalanguage – Extended BNF, <http://www.iso.ch/cate/d26153.html> (1996)
- International Organization for Standardization (ISO), ISO8879: Information processing – Text and office systems – Standard Generalized Markup Language (SGML), <http://www.iso.ch/cate/d16387.html> (1986)
- Ioannidis, Y.E.: Query optimization. In: *ACM Computing Surveys*. Vol. 28, No. 1 (1996)

- Jagadish, H.V., Mumick, I.S., Silberschatz, A.: View maintenance issues for the chronicle data model. In: Proceedings ACM symposium on principles of database systems (PODS 1995). pp. 113–124. (1995)
- Jarke, M., Koch, J.: Query optimization in database systems. In: ACM computing surveys. **16**(2) (1984)
- Jarrar, M., Dikaiakos, M.D.: MashQL: A query-by-diagram topping SPARQL – towards semantic data mashups. In: ONISW'08, Napa Valley, California, USA (2008)
- Jarrar, M., Dikaiakos, M.D.: A data mashup language for the data web, Linked Data on the Web (LDOW) at WWW. Madrid, Spain (2009)
- Jeon, M., Kim, D.: Load-balanced parallel merge sort on distributed memory parallel computers. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02). (2002)
- Kay, M.H.: Saxon – The XSLT and XQuery processor, <http://saxon.sourceforge.net> (2010)
- Kempa, M., Linnemann, V.: Type Checking in XOBEL, BTW 2003, Leipzig, Germany (2003)
- Kiefer, M.: Rule interchange format: The framework, Joint Keynote between RR2008 and RuleML-2008, slides available at <http://intranet.cs.man.ac.uk/ruleML/presentations/keynote3.ppt> (2008)
- Kifer, M., Lozinskii, E.L.: On compile-time query optimization in deductive databases by means of static filtering. ACM Trans Datab Syst **15**(3), 385–426 (1990)
- Kim, Y., Kim, B., Lee, J., Lim, H.: The path index for query processing on RDF and RDF Schema. ICACT. (2005)
- Kiryakov, A., Ognyanov, D., Kirov, V.: An ontology representation and data integration (ORDI) Framework, WP2 of DIP Project, 2004. <https://bscw.dip.deri.ie/bscw/bscw.cgi/0/3012>
- Kiryakov, A., Ognyanov, D., Manov, D.: OWLIM – a Pragmatic Semantic Repository for OWL. In: Proceedings of the International Workshop on Scalable SemanticWeb Knowledge Base Systems (SSWS'05), New York City, USA (2005)
- Kitsuregawa, M., Tanaka, H., Moto-oka, T.: Application of hash to data base machine and its architecture. New Generation Computing **1**(1) (1983)
- Kitsuregawa, M., Ogawa, Y.: A new parallel Hash join method with robustness for data skew in super database computer (SDC), In: Proceedings of the Sixteenth International Conference on Very Large Data Bases. Melbourne, Australia, August (1990)
- Kjernsmo, K., Passant, A.: SPARQL new features and rationale, W3C Working Draft 2 July 2009. Available at <http://www.w3.org/TR/2009/WD-sparql-features-20090702/>
- Klein, N., Groppe, S., Böttcher, S.: A prototype for translating XQuery expressions into XSLT stylesheets. In: Ninth East-European Conference on Advances in Databases and Information Systems (ADBIS 2005), Tallinn, Estonia (2005)
- Klug, A.: On conjunctive queries containing inequalities, J ACM, **35**(1) (1988)
- Knuth, D.E.: Sorting and searching, vol. 3 of The art of computer programming, 2nd edn. Reading, MA: Addison-Wesley (1998)
- Kopena, J.: OWLJessKB: A semantic web reasoning tool, <http://edge.cs.drexel.edu/assemblies/software/owljesskb/> (2005)
- Kruk, S.R., Decker, S.: Semantic social collaborative filtering with FOAFRealm. In: Proceedings of Semantic Desktop Workshop at ISWC 2005, Galway, Ireland (November 2005)
- Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'05). ACM, New York, pp. 1–12. (2005)
- Law, Y.-N., Wang, H., Zaniolo, C.: Query Languages and Data Models for Database Sequences and Data Streams. In: Proceedings of International Conference on Very Large Data Bases (VLDB 2004), pp 492–503. (2004)
- Law, Y.-N., Zaniolo, C.: An adaptive nearest neighbor classification algorithm for data streams. In: Proceedings of European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005), pp. 108–120. (2005)

- Lechner, S., Preuner, G., Schrefl, M.: Translating XQuery into XSLT, In: ER 2001 Workshops, Yokohama, Japan (2001)
- Ley, M.: The DBLP computer science bibliography. <http://www.informatik.uni-trier.de/~ley/db/> (2010)
- Liarou, E., Idreos, S., Koubarakis, M.: Continuous RDF query processing over DHTs. In: ISWC. (2007)
- Linnepe, D., Groppe, S., Ingenerf, J.: Nutzung von MEDLINE und MeSH für das Benchmarking von RDF-Speichersystemen, 54. gmds-Jahrestagung 2009 – Deutsche Gesellschaft für Medizinische Informatik, Biometrie und Epidemiologie e.V., Essen, Germany (2009)
- Lisi, F.A.: Building rules on top of ontologies for the semantic web with inductive logic programming. *Theory and Practice of Logic Programming (TPLP)* **8**(3), 271–300 (2008)
- Liu, Y.A., Stoller, S.D.: From datalog rules to efficient programs with time and space guarantees. *ACM Trans Program Lang Syst* **31**(6) (2009)
- Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. *IEEE Trans Knowl Data Eng* **11**(4), 610–628 (1999)
- MacGregor, R.M., Ko, I.-Y.: Representing contextualized data using semantic web tools. In: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems. Sanibel Island, FL, USA, October (2003)
- Malhotra, A., Melton, J., Walsh, N., (eds): XQuery 1.0 and XPath 2.0 Functions and Operators, W3C Recommendation, 23 January (2007)
- Manolescu, I., Florescu, D., Kossmann, D.: Pushing XML Queries inside Relational Databases. INRIA, Rapport de recherche 4112, (2001)
- Matono, A., Yoshikawa, A.T., Uemura, S.: An indexing scheme for RDF and RDF schema based on Suffix Arrays. SWDB'03 co-located with VLDB 2003, Berlin (2003)
- Matono, A., Amagasa, T., Yoshikawa, M., Uemura, S.: A path-based relational RDF database. In: ADC, (2005)
- McBride, B.: Jena: A semantic web toolkit. *IEEE Internet Computing* **6**(6), 55–59 (2002)
- McKay, D.P., Shapiro, S.C.: Using active connection graphs for reasoning with recursive rules. In: International Joint Conference on Artificial Intelligence. pp. 368–374. (1981)
- Microsoft, Biztalk, <http://www.biztalk.org> (2009)
- Microsoft: SQL Server 2005 Express, <http://www.microsoft.com/sql/express> (2007)
- Miller, R., Pippenger, N., Rosenberg, A., Snyder, L.: Optimal 2–3 trees, IBM Research Rep. RC 6505, IBM Research Lab. Yorktown Heights, NY. (1977)
- Miller, L., Seaborne, A., Reggiori, A.: Three implementations of SquishQL, a simple RDF query language, ISWC2002, Chia, Sardinai, Italy (2002)
- Mishra, P., Eich, M.: Join processing in relational databases. *ACM Computing Surveys* **24**(1), 63–113 (1992)
- MIT Libraries Barton Catalog Data. <http://simile.mit.edu/rdf-test-data/barton/> (2007)
- Möller, K., Dragan, L., Ambrus, O., Handschuh, S.: A visual interface for building SPARQL queries in Konduit. In: 7th International Semantic Web Conference (ISWC 2008), Karlsruhe, Germany (2008)
- Möller, K., Handschuh, S., Trüg, S., Josan, L., Decker, S.: Visual programming for the semantic desktop with Konduit. In: Proceedings of the 5th European Semantic Web Conference (ESWC2008), Tenerife, Spain (2008)
- Motik, B., Studer, R.: KAON2 – A scalable reasoning tool for the semantic web. In: Proceedings of the 2nd European Semantic Web Conference (ESWC'05), Heraklion, Greece (2005)
- Motik, B., Patel-Schneider, P.F., Parsia, B.: OWL 2 Web ontology language structural specification and functional-style syntax, W3C Recommendation, 27 October 2009
- Munagala, K., Srivastava, U., Widom, J.: Optimization of continuous queries with shared expensive filters. In: Proceedings ACM International Symposium on Principles of Database Systems (PODS 2007). pp. 215–224. (2007)
- Muñoz, S., Pérez, J., Gutierrez, C.: Minimal deductive systems for RDF. 4th European Semantic Web Conference (ESWC 2007), Innsbruck, Austria (2007)

- Nagy, L., Stansifer, R.: Polymorphic type inference for the relational algebra in the functional database programming language neon. In: 44th annual Southeast regional conference, Melbourne, USA (2006)
- Naur, P. (ed): Revised report on the algorithmic language ALGOL 60. *Computer J*, **5**(4), 349–367 (1963)
- Nejdl, W., Wolf, B., Qu, C., Decker, S., Naeve, MSA, Nilsson, M., Palmer, M., Risch, T., EDUTELLA: A P2P networking infrastructure based on RDF, WWW2002, Honolulu, Hawaii, USA (2002)
- Neumann, T., Weikum, G.: Scalable join processing on very large RDF graphs, SIGMOD (2009)
- Neumann, T., Weikum, G.: RDF3X: a RISCstyle engine for RDF. In: Proceedings of the 34th International Conference on Very Large Data Bases (VLDB). Auckland, New Zealand (2008)
- Nodine, M.H., Vitter, J.S.: Deterministic distribution sort in shared and distributed memory multiprocessors. In: Proceedings of the ACM Symposium on Parallel Algorithms and Architectures. Velen, Germany (1993)
- Northrop Grumman Corporation, Kowari, <http://kowari.sourceforge.net/> (2007)
- OpenLink iSPARQL, <http://demo.openlinksw.com/isparyl> (2010)
- Oracle.: Semantic Technologies Center, http://www.oracle.com/technology/tech/semantic_technologies/index.html (2009)
- Pan, Z., Heflin, J.: DLDB: Extending relational databases to support Semantic Web queries. In: PSSS. (2003)
- Parent, C., Spaccapietra, S.: About complex entities, complex objects and object-oriented data models. In: Proceedings of the IFIP 8.1 conference (1989)
- Paschke, A., Hirtle, D., Ginsberg, A., Patranjan, P.-L., McCabe, F.: RIF use cases and requirements. W3C Working Draft. 18 December 2008, <http://www.w3.org/TR/rif-ucr/>
- Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ISWC, Athens, USA (2006)
- Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* **34**(3) (2009)
- Peterson, D., Gao, S., Malhotra, A., Sperberg-McQueen, C.M., Thompson, H.S., (eds): W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. W3C Candidate Recommendation. <http://www.w3.org/TR/2009/CR-xsd-schema11-2-20090430/>, 30 April 2009. Latest version available as <http://www.w3.org/TR/xmlschema11-2/>
- Piatetsky-Shapiro, G., Connell, C.: Accurate estimation of the number of tuples satisfying a condition. SIGMOD, (1984)
- Polleres, A.: From SPARQL to rules (and back). In: WWW2007, Banff, Canada (2007)
- Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF, W3C Recommendation, (2008)
- Russell, A., Smart, P.R.: NITELIGHT: A graphical editor for SPARQL Queries. In: 7th International Semantic Web Conference (ISWC 2008), Karlsruhe, Germany (2008)
- Sacca, D., Zaniolo, C.: On the implementation of the simple class of logic queries for databases. In: Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems. ACM, New York, pp. 16–23. (1986)
- Schmidt, J.W.: Some high level language constructs for data of type relation. *ACM ToDS* **2**(3), 247–261 (1977)
- Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP²Bench: A SPARQL performance benchmark, ICDE. Shanghai, China (2009)
- Schneider, D., DeWitt D.: A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment, In: Proceedings of the 1989 SIGMOD Conference. Portland, OR, June 1989
- Schneider, D., DeWitt, D.: Tradeoffs in processing complex join queries via Hashing in multiprocessor database machines. In: Proceedings of the Sixteenth International Conference on Very Large Data Bases, Melbourne, Australia, August (1990)

- Schuhart, H., Linnemann, V.: Valid updates for persistent XML objects, BTW 2005, Karlsruhe, Germany (2005)
- Seaborne, A.: RDQL – A query language for RDF. W3C Member Submission, W3C. Available at: <http://www.w3.org/Submission/RDQL/> (2004)
- Seaborne, A., Manjunath, G.: SPARQL/Update, A language for updating RDF graphs. <http://jena.hpl.hp.com/~afs/SPARQL-Update.html> (2008)
- Semantic web challenge 2009. billion triples track. <http://challenge.semanticweb.org/>
- Semantic web challenge 2010. Billion triples track. <http://challenge.semanticweb.org/>
- Serfiotis, G., Koffina, I., Christophides, V., Tannen, V.: Containment and minimization of RDF/S query patterns. In: ISWC, Galway, Ireland (2005)
- Shadbolt, N.: The AKT project, <http://www.aktors.org/akt> (2007)
- Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D.J., Naughton, J.F.: Relational databases for querying XML documents: limitations and opportunities. VLDB 1999, Edinburgh, Scotland (1999)
- Shields, M., Meijer, E.: Type-indexed rows. In: PoPL, London, Great Britain (2001)
- Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL DL reasoner. J Web Semantics (2006)
- Smart, P.R., Russel, A., Braines, D., Kalfoglou, Y., Bao, J., Shadbolt, N.R.: A visual approach to semantic query design using a web-based graphical query designer. In: 16th International Conference on Knowledge Engineering: Practice and Patterns (EKW), Acitrezza, Italy (2008)
- Software AG, Tamino XML Server, <http://www.softwareag.com/tamino> (2007)
- SPARQLMotion <http://www.topquadrant.com/sparqlmotion> (2008)
- Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S.R., O’Neil, E.J., O’Neil, P.E., Rasin, A., Tran, N., Zdonik, S.B.: C-store: a column-oriented DBMS. In: VLDB, (2005)
- Subramanyam, G.V., Kumar, P.S.: Efficient handling of sibling axis in XPath, COMAD 2005, Goa, India (2005)
- Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: a core of semantic knowledge. In: WWW (2007)
- Swiss Institute of Bioinformatics, uniprot RDF, <http://dev.isb-sib.ch/projects/uniprot-rdf/> (2009)
- Sybase. SQLJ Part 1: Java Stored Procedures. Working Draft. (1998)
- Sybase. SQLJ Part 2: Java Data Types. Working Draft. (1998)
- Tamaki, H., Sato, T.: OLD resolution with tabulation. In: Shapiro, E. (ed.) Proceedings of the 3rd International Conference on Logic Programming, pp. 84–98. Springer, Berlin (1986)
- Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered XML using a relational database system. SIGMOD Conference 2002, Madison, Wisconsin, USA (2002)
- Tsarkov, D., Horrocks, I.: FaCT++ Description logic reasoner: system description. In: Proceedings of the International Joint Conference on Automated Reasoning (IJCAR’06), (2006)
- Tummarello, G., Polleres, A., Morbidoni, C.: Who the FOAF knows Alice? A needed step toward Semantic Web Pipes. In: Proceedings of the ISWC Workshops (2007)
- UMBC, Swoogle Semantic Web Search Engine, <http://swoogle.umbc.edu/> (2009)
- Uniprot RDF. <http://dev.isb-sib.ch/projects/uniprot-rdf/> (2009)
- van Assem, M., Gangemi, A., Schreiber, G.: RDF/OWL Representation of WordNet, W3C Working Draft, 2006. <http://www.w3.org/TR/wordnet-rdf/>
- van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J ACM **38**(3), 620–650 (1991)
- Vdovjak, R., Barna, P., Houben, G.: EROS: explorer for RDFS-based ontologies. In: Proceedings of Intelligent User Interfaces. Miami, FL, USA (2003)
- Volz, R., Oberle, D., Staab, S., Motik, B.: KAON SERVER - A Semantic Web Management System. In: WWW (2003)
- Walavalkar, O., Joshi, A., Finin, T., Yesha, Y.: Streaming knowledge bases. In: Proceedings of International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2008), (2008)

- Wallace, M., Runciman, C.: Haskell, and XML: Generic combinators or type-based translation? In: ICFP'99 (1999)
- Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. VLDB (2008)
- Weithöner T., Liebig T., Luther M., Böhm S., von Henke F., Noppens, O.: Real-world reasoning with OWL, ESWC 2007, Springer LNCS 4519. pp. 296–310. (2007)
- Wilkinson K.: Jena property table implementation. In: SSWS (2006)
- Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF Storage and Retrieval in Jena2. In: Workshop on Semantic Web and Databases. Berlin, Germany (2003)
- Williams, J.W.J.: Algorithm 232: Heapsort. Commun ACM 7(6), 347–348 (1964)
- Wolf, J.L., Dias, D.M., Yu P.S.: An effective algorithm for parallelizing sort-merge joins in the presence of data Skew. In: 2nd International Symposium on Databases in Parallel and Distributed Systems. (1990)
- Wood, D., Gearon, P., Adams, T.: Kowari: A platform for Semantic Web storage and analysis. In: XTech, (2005)
- World Wide Web Consortium (W3C), Document Object Model (DOM) Level 3 Core Specification Version 1.0, W3C Recommendation, <http://www.w3.org/TR/DOM-Level-3-Core/> (2004)
- World Wide Web Consortium (W3C), Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, <http://www.w3.org/TR/2004/REC-xml-20040204/>, February (2004)
- World Wide Web Consortium (W3C), XPath Version 2.0, W3C Recommendation. (2007)
- World Wide Web Consortium (W3C), XQuery 1.0: An XML Query Language, W3C Recommendation (2007)
- World Wide Web Consortium (W3C), XSL Transformations (XSLT) Version 2.0, W3C Recommendation, (2007)
- World Wide Web Consortium (W3C), Semantic Web, <http://www.w3.org/standards/semanticweb/> (2010)
- World Wide Web Consortium (W3C), XML Technology, <http://www.w3.org/standards/xml/> (2010)
- xmlmind, Qizx, <http://www.xmlmind.com/qizx/> (2010)
- Yoshikawa, M., Amagasa, T., Shimura, T., Uemura, S.: XRel: A path-based approach to storage and retrieval of XML documents using relational databases. ACM TOIT 1(2001), 110–141 (2001)
- Zeller, H.J., Gray, J.: Adaptive hash joins for a Multiprogramming Environment. In: Proceedings of the 1990 VLDB Conference. Australia, August 1990
- Zhao, J., Boley, H.: Uncertainty treatment in the rule interchange format: From encoding to extension. In: Fourth International Workshop on Uncertainty Reasoning for the Semantic Web. Karlsruhe, Germany (2008)
- Zhou, J., Ma, L., Liu, Q., Zhang, L., Yu, Y., Pan, Y.: Minerva: A scalable OWL ontology storage and inference system. In: Proceedings of 1st Asian Semantic Web Conference (ASWC 2006), Beijing, China (2006)
- Zloof, M.: Query-by-example: a data base language. IBM Syst J 16(4) (1977)

Index

A

Abstract syntax tree, 70
Access parallelism, 166
ACID, 165
Aggregation function, 18, 159
Algebra, 79
Amdahl's law, 167
ASC, 21
ASK, 19, 79
Atomicity, 165
Attribute grammar, 244

B

B⁺-tree, 3, 35, 36, 99, 103, 110, 112,
114, 126, 136, 170
Bag, 82
Barton dataset, 59
Basic Logic Dialect, 29
Batch speedup, 167
Billion Triples Challenge, 2, 7, 104, 146, 149, 173
Binding, 81
BizTalk, 226
Blank node, 10, 22, 79
Block-based nested loop join, 118
Bloom filter, 126
Bottom-up, 94
Bound, 23
Bounded buffer, 168
Branch and Bound, 95
Browser-like query creation, 191, 195
Bubble-down, 47
Bubble-up, 47
Built-in condition, 79
Built-in functions, 23
Bushy join tree, 91

C

Closed World Assumption, 16, 28
Cocoon, 226
Collation order, 104
Conceptual query languages, 193
Condensed data view, 196
Consistency, 165
Constant propagation, 88
CONSTRUCT, 19, 79
Continuous queries, 155
CoreSPARQL, 70, 73, 81
Cost estimation, 79
Cost-based optimization, 90

D

Data model, 221
Data parallelism, 166
Datatype, 23
DAWG test case, 235
DBpedia, 192
Default graph, 20
Deleting, 43
DESC, 21
DESCRIBE, 20
Dictionary indices, 110
Disk-based indices, 110
Distribution sort, 36, 52
Distribution Sort_{Component Keys}, 54
Distribution Sort for RDF, 35
Distribution Sort_{Triple Keys}, 54
Duplicate elimination, 105, 137
Durability, 165
Dynamic programming, 95
Dynamically restricting triple
patterns, 126

E

eBay, 155
 EBNF, 209
 Embedded language, 203
 Enumeration of plans, 94
 Equal-depth, 99
 Equal-width, 99
 Equi-depth, 113, 132, 170
 Equivalency rule, 79
 Evaluation indices, 111
 External chunks merge sort, 35, 50
 External merge sort, 36, 47
 External sorting, 3

F

False drops, 126
 Filter, 22
 First-order logic, 28
 FOAF, 104
 FROM, 20
 FROM NAMED, 20

G

Generate operator, 182
 Grammar, 70, 77, 244
 Graph pattern, 80, 88
 GROUP BY, 18

H

Hash join, 105, 123, 130, 136, 152
 Hash partitioning, 168
 Heap, 45
 Heuristic, 89
 Hill Climbing, 95
 Histogram, 93, 99, 113, 132, 170
 Histogram indices, 113
 Homomorphism, 211
 Horn logic, 28

I

Incompleteness, 7
 Inconsistency, 7
 Index construction, 35, 45, 68
 Index join, 122
 Inference, 3, 8, 177
 Inference rules, 178
 Infinite data streams, 155
 Initial run, 36
 In-memory indices, 109
 Inserting, 41
 Interior nodes, 36
 isBlank, 23

isIRI, 23
 isLiteral, 23
 Isolation, 165
 isURI *See* isIRI
 Iterator, 40, 118

J

Jena, 68
 Job parallelism, 166

K

k-chunks heap, 50

L

Leaves, 36
 Left outer-join, 83
 Left-deep join tree, 91
 LIMIT, 21
 Logical operators, 70
 Logic-based dialects, 28
 LUBM benchmark, 187
 LUPOSDATE, 3, 67, 86, 98, 126, 160, 193

M

Main memory database, 3, 91, 106, 139
 Materialization, 116, 179
 Materialization strategies, 3
 Merge join, 104, 120, 131, 152
 MergeOptional, 136
 Microsoft SQL Server, 2005 Express, 226
 Most-frequent-values, 99
 Multiple inheritance, 16
 Multiset *See* Bag
 Multi-user synchronization, 165

N

N3 notation, 11
 N3QL, 17
 Named graphs, 20, 80
 Negation as Failure, 23
 Nested subqueries, 18
 Nested-loop join, 117
 No Unique Name Assumption, 17
 Node splitting, 37
 Non-first-order logics, 28
 Notation, 3, 17

O

Object, 9, 104
 OFFSET, 21
 Ontology, 7, 9, 13, 28, 192, 208, 220
 Open World Assumption, 16, 28

Operatorgraph, 70, 138, 159, 179

Optimization

- join order, 91
- logical, 3, 70, 180
- logical optimization rule, 85
- physical, 3, 72, 103
- query, 79

OPTIONAL, 23, 136

Oracle, 17

ORDER BY, 21

P

Parallel database, 3, 163

Parallel monitor, 168

Partitioned parallelism, 164

Physical operator, 72, 103, 116, 138, 229

Pipeline-breaker, 116

Pipelining, 116

Plan generator, 113

Precompiler, 204

Predicate, 9, 104

Prefix key, 39

Prefix Search, 39

Presorting number, 131, 138

Production Rule Dialect, 29

Production rule systems, 28

Prolog, 30

Q

Qizx, 226

Query

- textual, 3
- visual, 3

Query head, 81

Query optimizer, 3

Query result type, 217

Query-by-example, 193

Query-by-form, 193

R

Range partitioning, 168

RDF Distribution Sort, 53

RDF Schema, 9, 14, 177, 208, 220

RDF stores, 27

RDF/XML, 13, 204

RDFa, 7

rdfDB, 17

RDQL, 17

Reactive (or event-condition-action)

- rules, 28

Refining queries, 197

REGEX, 23

Relational Database, 221

Replacement selection, 36, 48, 110

Resolution conflict strategy, 30

Resource Description Framework (RDF),
1, 7–9, 220

Restrictiveness, 91

RIF Core Dialect, 29

Right-deep join tree, 91

RSS 1.0, 104

Rule, 7, 9, 28, 177

Rule Interchange Format (RIF), 9, 28, 220

Rules-with-actions dialects, 28

S

Satisfiability test, 215

Saxon, 226

Search engines, 1

Searching, 39

SELECT, 18, 79

Self-balancing Property, 38

Selinger-Style, 97

Semantic Web, 1, 3, 7, 219

Semantic Web databases, 2, 163

Semantically duplicated terms, 7

Semantics

- stable, 28
- well-founded, 28

Semi-structured document, 221

Sesame, 17, 68

Sesame RDF Query Language (SeRQL), 17

Set-contains, 82

Set-difference, 82

Sideways Information Passing, 39, 116, 122,
125, 136

Social web, 191

Solution, 81

Sorting numbering scheme, 3, 105,
129, 145

SP²B benchmark, 57, 139

SPARQL, 1, 9, 17, 67, 79, 104, 163,
192, 204

Protocol, 24

Query Results, 26

SPARQL 1.1, 18, 24

SPARUL, 18, 204

SquishQL, 17

Standard Generalized Markup Language
(SGML), 219

Static program analysis, 204

Statistics-gathering cycles, 99

Stream operator, 160
Stream processing, 3, 68, 155
Subject, 9, 104
Subtype test, 210
SWOBE, 204

T

Tamino XML Server, 226
TAP, 17
Top-down, 94
Triple pattern, 18, 22
Type system, 208

U

Uncertainty, 7
UNION, 24
UniProt, 1, 7, 104, 146
Update, 18

V

Vagueness, 7
Variable, 21
Variable propagation, 87
Vastness, 7

Visual query language, 191
Visual Query System (VQS), 191

W

W3C test cases, 67, 112
Web Ontology Language (OWL), 9, 14,
177, 208, 220
WHERE, 18
Wikipedia, 2, 7, 58, 104, 192
Window, 155, 159, 160
WordNet, 2, 7, 58, 104
World Wide Web, 1

X

XML, 2, 3, 219, 221
DTD, 219
Schema, 219
XPath, 17, 219
XQuery, 17, 219
XSLT, 17, 219

Y

Yago dataset, 58
YARSQL, 17