



INTELLIGENT SYSTEMS

Barbara Catania
Lakhmi C. Jain (Eds.)

Editors-in-Chief

Prof. Janusz Kacprzyk
Systems Research Institute
Polish Academy of Sciences
ul. Newelska 6
01-447 Warsaw
Poland
E-mail: kacprzyk@ibspan.waw.pl

Prof. Lakhmi C. Jain
School of Electrical and Information
Engineering
University of South Australia
Adelaide
South Australia SA 5095
Australia
E-mail: Lakhmi.jain@unisa.edu.au

Barbara Catania and Lakhmi C. Jain (Eds.)

Advanced Query Processing

Volume 1: Issues and Trends

 Springer

Editors

Prof. Barbara Catania
Department of Computer and
Information Science
University of Genoa
Genoa
Italy

Prof. Lakhmi C. Jain
School of Electrical and
Information Engineering
University of South Australia
Adelaide, SA
Australia

ISSN 1868-4394

ISBN 978-3-642-28322-2

DOI 10.1007/978-3-642-28323-9

Springer Heidelberg New York Dordrecht London

e-ISSN 1868-4408

e-ISBN 978-3-642-28323-9

Library of Congress Control Number: 2012933077

© Springer-Verlag Berlin Heidelberg 2013

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, 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.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

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

Preface

The last decade has been characterized by the raise of data intensive applications, with new data querying needs and novel processing environments. Data integration applications, Web services, sensors networks, P2P, cloud computing, and hosting are only few examples of these emerging technologies. The novel processing requirements characterizing such new applications and environments made traditional query processing approaches unsatisfactory and required the development of specific advanced query processing techniques. The aim of this book is to present key developments, directions, and challenges concerning advanced query processing for both traditional (i.e., relational) and non-traditional data (like XML and spatial data), either stored or available as a stream. A special emphasis is devoted to approximation and adaptivity issues as well as to integration of heterogeneous data sources.

The book could be used as a reference book for senior undergraduate or graduate courses on advanced data management issues, which have a special focus on query processing and data integration. It is also useful for technologists, managers, and developers who want to know more about emerging trends in advanced query processing.

The book is organized into five parts. Each part deals with a specific advanced query processing issue and contains contributions of researchers, selected by the editors, which are experts in their respective research areas. The first part of the book consists of three chapters and deals with preference-based query processing for traditional relational data. The second part, composed of two chapters, deals with approximate query processing issues for non-traditional data, like XML and spatial data. The third part consists of two chapters dealing with continuous query processing in data stream management systems and data warehousing environments. The fourth part contains two chapters concerning adaptive query processing. Finally, the fifth part is composed of two chapters dealing with the integration of heterogeneous data sources and related querying issues.

The editors would like to thank all authors of this book for their insights and excellent contributions to this book. It was a great honour to collaborate with this team of very talented experts. Most of them also served as referees for chapters written by other authors. We wish to thank all of them, as well as all the other reviewers, for their constructive and comprehensive reviews. Special thanks are due to the publishing team at Springer, for its valuable assistance during the preparation of this manuscript.

October 2011

Barbara Catania
Italy

Lakhmi C. Jain
Australia

Contents

1	Advanced Query Processing: An Introduction	1
	<i>Barbara Catania, Lakhmi Jain</i>	
1.1	Introduction	1
1.2	Preference-Based Query Processing	5
1.3	Approximate Query Processing for Non-traditional Data	6
1.4	Continuous Query Processing	8
1.5	Adaptive Query Processing	9
1.6	Queries over Heterogeneous Data Sources	10
1.7	Conclusion and Discussion	11
	References	12

Part I

2	On Skyline Queries and How to Choose from Pareto Sets	15
	<i>Christoph Lofi, Wolf-Tilo Balke</i>	
2.1	Introduction	15
2.2	Formalization of Skyline Sets Following the Pareto Semantics	17
2.3	Relaxing the Pareto Semantics	21
2.4	Summarizing the Skyline	23
	2.4.1 Approximately Dominating Representatives	23
	2.4.2 Statistical Sampling Skylines	24
2.5	Weighting Characteristics of Skyline Points	25
	2.5.1 Skycubes and Subspace Analysis	26
	2.5.2 SKYRANK	28
	2.5.3 k Most Representative Skyline Points	29
	2.5.4 Personalized Top-k Retrieval / Telescope	30
2.6	Cooperative Approaches	31
	2.6.1 Interactive Preference Elicitation	31
	2.6.2 Trade-Off Skylines	32
2.7	Conclusion and Discussion	33
	References	34

3	Processing Framework for Ranking and Skyline Queries	37
	<i>Seung-won Hwang</i>	
3.1	Introduction	37
3.2	Related Work	40
3.2.1	Sorted-Order Algorithms	40
3.2.2	Partitioning-Based Algorithms	41
3.2.3	Taxonomy and Generalization	41
3.3	Framework for Ranking Queries	42
3.3.1	Preliminaries	42
3.3.2	Extracting Skeleton	44
3.3.3	Cost-Based Optimization Strategies	45
3.4	Framework for Skyline Queries	47
3.4.1	Preliminaries	47
3.4.2	Extracting Skeleton	49
3.4.3	Cost-Based Optimization Strategies	53
3.5	Conclusion and Open Issues	55
	References	55
4	Preference-Based Query Personalization	57
	<i>Georgia Koutrika, Evaggelia Pitoura, Kostas Stefanidis</i>	
4.1	Introduction	57
4.2	Preference Representation	59
4.2.1	Context Specification	59
4.2.2	Preference Specification	61
4.2.3	Combining Preferences	64
4.2.4	Example: A User Profile	66
4.3	Personalizing Queries Using Preferences	68
4.3.1	Preference Selection	68
4.3.2	Personalized Query Processing	73
4.4	Preference Learning	76
4.5	Conclusion and Open Issues	77
	References	78
Part II		
5	Approximate Queries for Spatial Data	83
	<i>Alberto Belussi, Barbara Catania, Sara Migliorini</i>	
5.1	Introduction	83
5.2	Background on Spatial Data and Queries	86
5.3	A Taxonomy of Query-Based Approximation Techniques for Spatial Data	89
5.3.1	Query Relaxation	89
5.3.2	Approximate Query Processing	92

5.4	Spatial Top- <i>k</i> Queries	93
5.4.1	Top- <i>k</i> Ranking Function	94
5.4.2	Top- <i>k</i> Query Processing Algorithms	96
5.5	Spatial Skyline Queries	104
5.5.1	Spatial Skyline Queries	105
5.5.2	Spatial Skyline Query Processing Algorithms	106
5.6	Approximate Query Processing	112
5.6.1	Approximate Algorithms for Multiway Spatial Join	113
5.6.2	Approximate Algorithms for Distance-Based Queries	115
5.6.3	Algorithms Based on Approximate Spatial Data	116
5.7	Towards Qualitative Approximation Techniques for Spatial Data	118
5.7.1	From Qualitative to Quantitative Spatial Relations	119
5.7.2	Spatial Top- <i>k</i> Queries Based on Qualitative Relations	121
5.7.3	Spatial Skyline Queries Based on Qualitative Relations	123
5.8	Conclusion and Open Issues	124
	References	124
6	Approximate XML Query Processing	129
	<i>Giovanna Guerrini</i>	
6.1	Introduction	129
6.2	Twig Queries	132
6.2.1	XML Documents	133
6.2.2	Twig Queries: Definition	134
6.2.3	(Exact) Twig Query Processing	135
6.2.4	Twig Queries as a Basis for Approximate Querying	136
6.3	Various Extents of Approximation	138
6.3.1	Vocabulary	138
6.3.2	Hierarchical Structure	140
6.4	Ranking	142
6.4.1	Tree Edit Distance	143
6.4.2	An Alternative Match Based Similarity	144
6.4.3	Structure and Content <i>tf · idf</i> Scoring	145
6.4.4	Content Scoring with Structure Filters	146
6.5	Approximate Query Processing	148
6.5.1	Twig-Path Scoring and Whirpool	148
6.5.2	TopX	149

6.5.3	TASM	150
6.5.4	ArHeX	150
6.6	Conclusion and Discussion	151
	References	153

Part III

7	Progressive and Approximate Join Algorithms on Data Streams ...	157
	<i>Wee Hyong Tok, Stéphane Bressan</i>	
7.1	Introduction	157
7.2	Background	158
7.3	Why Progressive Joins?	160
7.4	Joins from Different Data Models Flock Together	161
7.4.1	Relational Joins	161
7.4.2	Spatial Joins	163
7.4.3	High-Dimensional Distance-Similarity Joins	164
7.4.4	Progressive XML Structural Joins	164
7.5	Generic Progressive Join Framework	164
7.5.1	Building Blocks for Generic Progressive Join Framework	165
7.5.2	Progressive Join Framework	166
7.5.3	RRPJ Instantiations	170
7.6	Progressive Approximate Joins	170
7.6.1	Extreme Scenario	171
7.6.2	Measuring the Performance of Progressive, Approximate Joins	172
7.6.3	Different Types of Progressive, Approximate Joins	174
7.6.4	Discussion	180
7.7	Open Issues	181
7.8	Conclusion	182
	References	183
8	Online Aggregation	187
	<i>Sai Wu, Beng Chin Ooi, Kian-Lee Tan</i>	
8.1	Introduction	187
8.2	Basic Principles	190
8.2.1	Statistical Model	192
8.2.2	Sampling	193
8.3	Advanced Applications	194
8.3.1	Online Aggregation for Multi-relation Query Processing	194
8.3.2	Online Aggregation for Multi-query Processing	197

8.3.3	Distributed Online Aggregation	203
8.3.4	Online Aggregation and MapReduce	207
8.4	Conclusion and Discussion	209
	References	209

Part IV

9	Adaptive Query Processing in Distributed Settings	211
	<i>Anastasios Gounaris, Efthymia Tsamoura, Yannis Manolopoulos</i>	
9.1	Introduction	211
9.1.1	Distributed Query Processing Basics	213
9.1.2	Related Work	214
9.2	A Framework for Analysis of AdQP	214
9.3	AdQP in Centralized Settings	215
9.3.1	Overview of Techniques	215
9.3.2	On Applying Conventional AdQP Techniques in Distributed Settings	216
9.4	AdQP for Distributed Settings: Extensions to Eddies	217
9.4.1	Techniques	218
9.4.2	Summary	220
9.5	AdQP for Distributed Settings: Operator Load Management	220
9.5.1	Intra-Operator Load Management	221
9.5.2	Inter-Operator Load Management	226
9.5.3	More Generic Solutions	230
9.6	AdQP for Distributed Settings: Other Techniques	230
9.7	Conclusion and Open Issues	232
	References	233
10	Approximate Queries with Adaptive Processing	237
	<i>Barbara Catania, Giovanna Guerrini</i>	
10.1	Introduction	237
10.2	QoD ² Techniques: Some Examples	241
10.2.1	Adaptively Approximate Pipelined Joins	241
10.2.2	Adaptive Processing of Skyline-Based Queries over Data Streams	243
10.3	QoD-Oriented Approximate Queries	244
10.3.1	Query Rewriting	245
10.3.2	Preference-Based Queries	246
10.3.3	Approximate Query Processing	248
10.4	QoS-Oriented Approximate Queries	249
10.4.1	Data Reduction	250
10.4.2	Load Shedding	253
10.4.3	Approximation of the Processing Algorithm	254

10.5	Adaptive Query Processing	254
10.5.1	Adapting Query Plans	255
10.5.2	Adaptively Coping with Limited Resources under Fixed Plans	259
10.5.3	Further Adaptation Subjects	260
10.6	Conclusion and Discussion	262
	References	263

Part V

11	Querying Conflicting Web Data Sources	271
	<i>Gilles Nachouki, Mohamed Quafafou, Omar Boucelma, François-Marie Colonna</i>	
11.1	Introduction	272
11.2	Conflicting Web Data Sources	274
11.2.1	Overview of Conflict Types	274
11.2.2	Conflicting Data in Life Sciences	275
11.2.3	Assumptions about Conflict Representation	277
11.3	Mediating Biological Conflicting Data with BGLAV	277
11.3.1	BGLAV Overview	277
11.3.2	Query Processing in BGLAV	278
11.4	MFA - Multi-source Fusion Approach	281
11.4.1	MFA Overview	281
11.4.2	Methodology for Semantic Reconciliation	285
11.4.3	Query Processing in MFA	287
11.5	Application	294
11.5.1	Data Source Description	295
11.5.2	BGLAV Illustrating Examples	295
11.5.3	MFA Illustrating Examples	297
11.5.4	Evaluation of MFA Queries	300
11.6	Conclusion and Open Issues	301
	References	301
12	A Functional Model for Dataspace Management Systems	305
	<i>Cornelia Hedeler, Alvaro A.A. Fernandes, Khalid Belhajjame, Lu Mao, Chenjuan Guo, Norman W. Paton, Suzanne M. Embury</i>	
12.1	Introduction	305
12.2	Dataspace Life Cycle	307
12.3	Background	309
12.3.1	Dataspace Management Systems	310
12.3.2	Model Management Systems	312
12.4	Functional Model	313
12.4.1	An Overview	313
12.4.2	Preliminary Assumptions	314
12.4.3	Intensional Descriptions	315
12.4.4	Sorts	318

12.4.5	Operations	321
12.5	Bioinformatics Use Case	329
12.5.1	Example: Dataspace Initialization	329
12.5.2	Example: Dataspace Maintenance	331
12.5.3	Example: Dataspace Improvement	334
12.6	Conclusion and Open Issues	336
	References	337
Author Index		343
Subject Index		345
Editors		349

List of Contributors

Wolf-Tilo Balke

Institute for Information Systems
Technische Universität Braunschweig
Mühlenpfordtstr. 23
D-38106 Braunschweig - Germany
balke@ifis.cs.tu-bs.de

Khalid Belhajjame

School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL - UK
khalidb@cs.manchester.ac.uk

Alberto Belussi

Department of Computer Science
University of Verona
Strada le Grazie, 15
Verona, 37134 - Italy
alberto.belussi@univr.it

Omar Boucelma

LSIS-UMR CNRS 6168
Aix-Marseille University
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 - France
Omar.Boucelma@lisis.org

Stéphane Bressan

School of Computing
National University of Singapore
13 Computing Drive
Singapore 117417
Republic of Singapore
steph@comp.nus.edu.sg

Barbara Catania

Department of Computer and
Information Science
University of Genoa
Via Dodecaneso, 35
Genoa, 16146 - Italy
barbara.catania@unige.it

François-Marie Colonna

Institut Supérieur de l'Electronique
et du Numérique
Maison des technologies
Place Georges Pompidou
83000 Toulon - France
francois-marie.colonna@isen.fr

Suzanne M. Embury

School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL - UK
sembury@cs.manchester.ac.uk

Alvaro A. A. Fernandes

School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL - UK
a.fernandes@cs.manchester.ac.uk

Anastasios Gounaris

Department of Informatics
Aristotle University
54124 Thessaloniki - Greece
gounaria@csd.auth.gr

Giovanna Guerrini

Department of Computer and
Information Science
University of Genoa
Via Dodecaneso, 35
Genoa, 16146 - Italy
giovanna.guerrini@unige.it

Chenjuan Guo

School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL - UK
guoc@cs.manchester.ac.uk

Cornelia Hedeler

School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL - UK
chedeler@cs.manchester.ac.uk

Seung-won Hwang

Department of Computer Science &
Engineering, POSTECH
790-784 San 31, Hyoja-dong,
Nam-gu, Pohang, Gyoungbuk
Republic of Korea
swhwang@postech.edu

Lakhmi C. Jain

School of Electrical and Information
Engineering
University of South Australia
Adelaide, Mawson Lakes Campus
South Australia SA 5095 - Australia
Lakhmi.Jain@unisa.edu.au

Georgia Koutrika

IBM Almaden Research Center
650 Harry Road
San Jose, California 95120-6099
USA
gkoutri@us.ibm.com

Christoph Lofi

Institute for Information Systems
Technische Universität Braunschweig
Mühlenpfordtstr. 23
D-38106 Braunschweig - Germany
lofi@ifis.cs.tu-bs.de

Yannis Manolopoulos

Department of Informatics
Aristotle University
54124 Thessaloniki - Greece
manolopo@csd.auth.gr

Lu Mao

School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL - UK
maol@cs.manchester.ac.uk

Sara Migliorini

Department of Computer Science
University of Verona
Strada le Grazie, 15
Verona, 37100 - Italy
sara.migliorini@univr.it

Gilles Nachouki

LINA-UMR CNRS 6241
Nantes University
2, rue de la Houssinière
F-44322 Nantes Cedex 03 - France
Gilles.Nachouki@univ-nantes.fr

Beng Chin Ooi

School of Computing
National University of Singapore
13 Computing Drive
Singapore 117417
Republic of Singapore
ooibc@comp.nus.edu.sg

Norman W. Paton

School of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL - UK
norm@cs.manchester.ac.uk

Evaggelia Pitoura

Computer Science Department
University of Ioannina
GR-45110 Ioannina - Greece
pitoura@cs.uoi.gr

Mohamed Quafafou

LSIS-UMR CNRS 6168
Aix-Marseille University
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 - France
Mohamed.Quafafou@lsis.org

Kostas Stefanidis

Department of Computer Science and
Engineering
Chinese University of Hong Kong
Sha Tin, New Territories, Hong Kong
SAR - China
kstef@cse.cuhk.edu.hk

Kian-Lee Tan

School of Computing
National University of Singapore

13 Computing Drive
Singapore 117417
Republic of Singapore
tank1@comp.nus.edu.sg

Wee-Hyong Tok

Microsoft China
Blk 635 Veerasamy Road
#10-162, Singapore (200635)
Republic of Singapore
weetok@microsoft.com

Efthymia Tsamoura

Department of Informatics
Aristotle University
54124 Thessaloniki - Greece
etsamour@csd.auth.gr

Sai Wu

School of Computing
National University of Singapore
13 Computing Drive
Singapore 117417
Republic of Singapore
wusai@comp.nus.edu.sg

Chapter 1

Advanced Query Processing: An Introduction

Barbara Catania and Lakhmi Jain

Abstract. Traditional query processing techniques have played a major role in the success of relational Database Management Systems over the last decade. However, they do not obviously extend to much more challenging, unorganized and unpredictable data providers, typical of emerging data intensive applications and novel processing environments. For them, advanced query processing and data integration approaches have been proposed with the aim of still guaranteeing an effective and efficient data access in such more complex data management scenarios. The aim of this chapter is to present the main issues and trends arising in advanced query processing and to relate them to the various parts of this book. For each part, a brief description of the background concepts and of the presented contributions is also provided.

1.1 Introduction

One of the main reasons for the success of Database Management Systems (DBMSs) and of the main key concepts upon which traditional query processing techniques have been developed is *logical data independence*. With logical data independence we mean the neat separation between the specification of ‘what’ we are searching for from ‘how’ these searches, specified in terms of queries, are processed. The system is responsible for transforming declarative queries into execution plans, statically determined before the processing starts. The result obtained by processing

Barbara Catania
University of Genoa, Italy
e-mail: barbara.catania@unige.it

Lakhmi Jain
University of South Australia, Australia
e-mail: Lakhmi.Jain@unisa.edu.au

the query according to the chosen execution plan is the set of items which exactly satisfy the specified query conditions.

Since the late 1970s and the introduction of System R [17], this approach has proved to be highly efficient and effective and it has played a major role in the relational DBMSs success over the last decades. Efficiency is guaranteed by the usage of several sophisticated optimization techniques. These techniques heavily rely on the existence of metadata information about the data which have to be processed, such as the distribution of values and the selectivity of the relational operators. Effectiveness is guaranteed by the usage of simple declarative languages which perfectly adhered to the needs of traditional database domains and applications, characterized by data which have a completely known structure, are executed in stable environments, and for which a reasonable set of statistical information on data is usually available.

The query processing language-to-engine stack of classical DBMSs embodies advanced and fundamentally elegant concepts and ideas, that however do not obviously extend to much more challenging, unorganized and unpredictable data providers, typical of emerging data intensive applications and novel processing environments. New applications and environments include, to mention only a few, data integration applications, web services, Future Web, sensors databases and networks, P2P, cloud computing, and hosting. They are characterized by: high network connectivity and resource sharing (as in data integration applications, Future Web, cloud computing, and hosting); new types of data availability (data can be stored or produced as a stream as in sensor databases); high data heterogeneity and incompleteness (as in data integration applications, P2P, and cloud computing); extremely high variability and unpredictability of data characteristics during the processing (because transactions could be long-running or the data schema may change during computation); limited user knowledge about the data which have to be processed and limited resources with respect to the data volumes under processing (the response time should be low also in presence of high volume of data as in cloud computing or sensor databases and the space may not be sufficient to store all of the data, which are sometimes unbounded, as in data streams).

All the characteristics discussed above make traditional query processing and data integration approaches not feasible for most of the new processing environments and lead to a radical modification of query processing requirements. As a consequence, new query processing approaches have been defined, that we call *advanced*. Two main innovative aspects are taken into account by advanced query processing techniques. A first issue concerns *approximation*. Data characteristics (e.g., heterogeneity, incompleteness, and uncertainty), resource limitations, huge data volumes, and volatility, typical of the new applications and processing environments, suggest it may be preferred to relax the query definition, using *Query Relaxation* (QR) techniques, or to generate an approximate result set, with quality guarantees, using *Approximate Query Processing* (ApQP) techniques, instead of getting an unsatisfactory answer. An answer can be unsatisfactory because either the user has

to wait too long for getting the result, the answer is empty, or the answer contains too many answers, not very significant for the user needs. Among QR techniques, *Preference-based Query Processing* (PQP) techniques take user preferences into account in relaxing the query. Preferences can be specified either in the query as soft constraint (*Preference-based Queries* (PQ)) or inside a user profile. On the other hand, ApQP approaches generate approximate query results by modifying either the data to be processed or the processing itself. In both cases, the approximation is finalized at improving either the quality of the result, in terms of Quality of Data parameters as completeness or accuracy, or the resource usage, in presence of limited or constrained resource availability, specified according to Quality of Service parameters.

A second issue concerns runtime *adaptation*. The instability of the new environments, due to the high network connectivity and heterogeneity of the shared resources, leads to unpredictability. This in turn leads to the impossibility of statically devising optimization strategies that will be optimal even beyond the very short term. At the same time, new processing environments have to deal not only with stored data but also with data continuously arriving as a stream (as in sensor networks). *Continuous Query Processing* (CQP) approaches have therefore been devised in order to be able to process a query in a continuous way along the time. For them, statically detected execution plans do not constitute a suitable approach; rather, the processing has to be adapted to dynamic conditions that may change during the query execution. This approach leads to the definition of *Adaptive Query Processing* (AdQP) techniques. AdQP techniques, which have been defined for both stored data, in centralized and distributed architectures, and data streams, revise the chosen execution plan on the fly, during processing, based on the changing conditions. For both ApQP and AdQP techniques, user interaction with the query processor becomes necessary to guide the processing towards the specific user needs.

Besides their application to stored data and data stream processing, approximation and adaptivity issues are now taken into consideration also for data integration of *Heterogeneous Data Sources* (HDS). This is useful in all situations in which data sources cannot be predefined but should be selected and combined on-demand, as it happens in mash-up applications [14].

This book addresses the topics mentioned above by giving contributions related to advanced query processing for both traditional (i.e., relational) and non-traditional data (like XML and spatial data), either stored or available as a stream. A special emphasis is devoted to approximation and adaptivity issues as well as to integration of heterogeneous data sources. More precisely, the book presents key developments, directions, and challenges related to preference-based query processing of stored data, preference-based and approximate query processing of non-traditional data, continuous query processing, adaptive query processing, and query processing on heterogeneous data sources. The book covers those issues and devotes a separate part of the book to each of them, according to the following organization:

- **PART 1** - Preference-based Query Processing.
- **PART 2** - Approximate Query Processing for Non-Traditional Data.

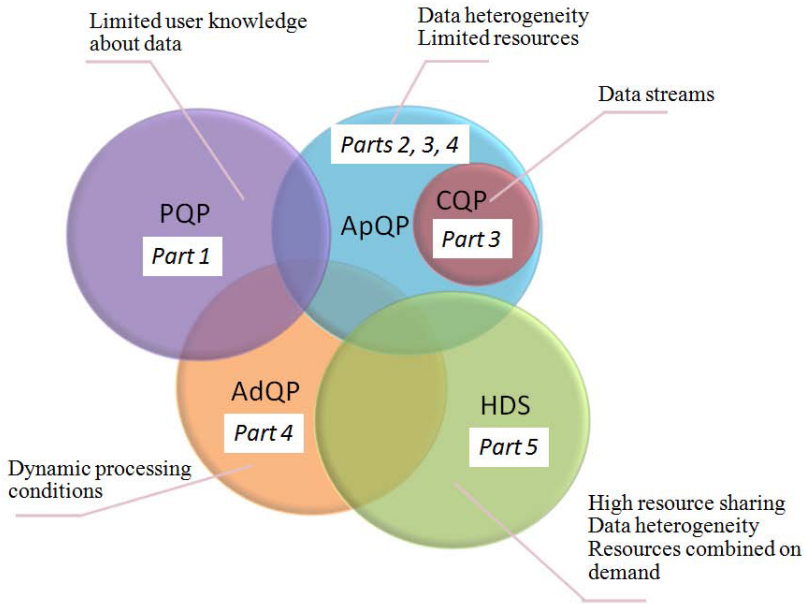


Fig. 1.1 An overall picture of the problems and techniques surveyed in this book.

- **PART 3** - Continuous Query Processing.
- **PART 4** - Adaptive Query Processing.
- **PART 5** - Queries over Heterogeneous Data Sources.

Figure 1.1 provides an outline of: (i) the query processing issues considered in this book; (ii) the problems addressed by each group of query processing techniques (limited user knowledge about data, data heterogeneity, limited resources, data streams processing, dynamic processing conditions, high resource sharing, and resources combined on demand); (iii) the parts of this book which cover each issue. In the figure, each issue is represented as a circle while addressed problems are connected by a line to the corresponding issue. We notice that the considered issues have been rarely investigated in isolation. Approximate query processing techniques have also been proposed in order to process preference-based queries. Adaptive solutions have been provided for approximate or preference-based techniques. Continuous query processing approaches are by definition approximate, in order to cope with the unboundness of data streams. Finally, techniques for the integration of heterogeneous data sources can be applied in an approximate and adaptive way, to cope with the heterogeneous nature of very dynamic environments (e.g., dataspaces [7, 9]).

In the following, we dedicate a section to each of the book parts, providing a brief description of the background concepts related to the presented contributions.

1.2 Preference-Based Query Processing

New processing environments, like P2P, cloud computing, Future Web, and sensor networks, to mention only a few, are characterized by the high heterogeneity and an extremely high variability and unpredictability of data characteristics, which often result in inconsistent and ambiguous datasets. Under these conditions, it is really difficult for the user to exactly specify what she looks for since, at query specification time, the user knowledge about data may be limited. This may happen even if data come from just one single source (possibly, because such characteristics may change during query execution, as in mash-up applications [14]) but the problem is more evident in distributed architectures, where input data may come from many different sources, with different formats. As a consequence, the quality of the obtained result, in terms of completeness and effectiveness, may decrease since interesting objects may not be returned (*empty answer problem*) [1]. On the other hand, several uninteresting answers may be returned, thus reducing user satisfaction (*many answer problem*) [1].

A typical approach to cope with the empty and many answer problems consists in involving the user in the query processing. This can be done by letting the user specify preferences instead of setting hard query constraints, as it happens in traditional query processing approaches. When executing a query based on some user preferences, the best query answers are returned with respect to the user preferences, even if they only partially satisfy the specified request, relying on preference-based query processing (PQP) techniques.

Preferences can be either specified inside the query as soft constraint, as it happens in preference-based queries (PQ), or collected in a user profile. In this second case, preferences can be used by the query processor to detect and return first the results assumed to be relevant for the user, based on the specified conditions (*query personalization* [13]). In both cases, two main categories of preferences can be devised: quantitative and qualitative. Quantitative preferences can be specified in terms of a ranking function which, on a database item, returns a numeric value representing the relevance of that item for the user. Quantitative preferences are often used for computing, through a *top-k query*, the best k items, i.e., the k items with the highest rank [12]. Alternatively, qualitative preferences are specified using binary predicates to compare items. A simple way to specify qualitative preferences is to first choose a set of interest attributes, then provide a comparison relationship for the domain of each attribute in this set, and finally consider the *dominance* relationship as the preference relationships between pairs of items [3]. An item A dominates an item B if A is better than B with respect to at least one attribute and it is at least equal to B with respect to all the other interest attributes. Qualitative preferences as the ones just described are typically used in *skyline queries*, which retrieve the set of items which are not dominated by any other item.

PQP is usually not an easy task. For what concerns top- k queries, the main issue is to find efficient algorithms which deal with the largest class of ranking functions and, at the same time, which avoid the computation of the ranks for all dataset items. While top- k operators return a small result at the price of specifying a ranking

function, skyline operators avoid this specification at the price of a larger result set. Even for two dimensional interest attributes, such result set may be quite large. In order to cope with such curse of dimensionality, specific solutions usually integrate both top- k and skyline advantages into a single technique. Additionally, depending on the type of the considered conditions and queries, query personalization techniques may be very expensive and optimized approaches need to be provided.

The aim of Part I, which consists of three chapters, is to discuss complementary issues and to present some solutions to PQP problems which have been discussed above, in the context of traditional relational data. The first two chapters deal with PQ while the third chapter deals with query personalization.

More precisely, Chapter 2 formally introduces skyline queries and focuses on drawbacks arising from their usage. A special emphasis is devoted to the curse of dimensionality problem. The chapter first classifies existing techniques to remedy this problem, based on the selection of the most interesting objects from the multitude of skyline objects in order to obtain truly manageable and personalized query results; relevant proposals for each identified class are then described in details.

Skyline queries, together with top- k queries, constitute the topic of Chapter 3. Existing algorithms for top- k and skyline queries are first considered with the aim of providing a meta-algorithm framework for each query type. The proposed framework generalizes existing algorithms and it allows the observation of important principles, which cannot be observed from the analysis of individual algorithms.

Finally, Chapter 4 deals with query personalization methods. The considered techniques rely on quantitative user preferences provided as a user profile separately from the query and dynamically determine how those profiles will affect the query results. A detailed discussion concerning how preferences can be represented and stored in user profiles and how preferences are selected from a user profile and then applied to a query is provided.

1.3 Approximate Query Processing for Non-traditional Data

Spatial and XML data play a relevant role in new application environments, the first for the diffusion of geo-referenced data on local and distributed applications, the second being a de-facto standard for data representation and transmission over the Web.

While spatial data are characterized by a more complex structure with respect to relational data, XML documents are semi-structured. This means that structure, i.e., schema information, is only partially represented inside them. Peculiarities of both data models require a revision of traditional and advanced query processing techniques to meet the new data model features and processing requirements.

With respect to traditional, non geo-referenced data, spatial data are characterized by an intrinsic complexity. This, from the very beginning, has required the

usage of specific query processing techniques, based on approximation. In the first Geographic Information Systems (GISs) [16], approximation mainly concerned data capture and data representation, that is, the accuracy concerning absolute and relative positions of objects in the embedded space which directly influences the accuracy of query results. Query processing techniques for spatial data usually require the approximation of arbitrary geometric data to simpler objects, such as rectangles or convex polygons [16]. Such simpler objects are then indexed and used to answer queries under a filtering-refinement approach.

When considering XML documents, traditional queries are expressed through standard XML query languages, such as XPath [19] and XQuery [20]. These languages extend relational languages to a tree-based structure, typical of XML data, and are built on the assumption of a regular structure with well-defined parent/child relationships between the nodes. They allow one to specify conditions over specific parts of the XML trees and return as query result a new XML document, generated starting from the subtrees of the original XML tree which satisfy the specified conditions.

Traditional languages and query processing approaches for spatial and XML data are not sufficient to cope with data intensive applications and advanced processing environments. Solutions to the empty and the many answer problems, such as those introduced in Section 1.2 and presented in Part 1 of this book, and ApQP techniques have been provided for non-traditional data, possibly revising and extending solutions already defined for relational data. For spatial data, techniques for relaxing or approximating spatial queries to be executed, instead of data to be processed, have been proposed. For XML data, solutions to relax content and structure conditions in queries on XML document collections and approximate processing techniques that are tolerant to significant variability in document structures have been provided.

Part II consists of two chapters presenting issues concerning PQ and ApQP techniques for stored spatial data and XML documents, respectively.

Approximation techniques for spatial data are the topic of Chapter 5. The chapter focuses on second-generation approximation techniques (either based on QR or ApQP approaches) which, differently from traditional approximation approaches for spatial data approximate the query result to be produced and not the data to be processed. Such techniques are first surveyed and then the issues that need further investigation are pointed out. Among them, a special emphasis is given to new types of preference-based queries relying on qualitative relations (namely, topological, cardinal, and distance-based relations).

Issues concerning query relaxation for XML documents are the topic of Chapter 6. Approaches to relax both content and structure conditions in queries on XML document collections and to rank results according to some similarity measure, as well as top- k processing approaches to efficiently evaluate them, are first classified and alternative solutions are discussed in details.

1.4 Continuous Query Processing

In traditional application environments, data is first gathered and then stored, in either centralized or distributed architectures. However, the ubiquitous network connectivity, characterizing most of the new application environments, allows data to be delivered as a *stream*. A stream is a continuous, potentially unbounded, voluminous, real-time, sequence of data elements, instances of a variety of data models (e.g., relational, spatial, high-dimensional, XML). Each data item in a stream usually represents either a notification that an interaction between entities, e.g., a credit card purchase, has taken place (transactional stream) or a notification that some entity has changed, e.g., the temperature of a room has changed (monitoring stream).

In order to process data streams, besides traditional one-time queries, which are evaluated once over a point-in-time snapshot of the dataset, with the answer returned to the user, *continuous* (also called *progressive*) queries become relevant. Continuous queries are evaluated continuously as data streams continue to arrive; query results are then continuously updated (e.g., for aggregates) or produced as a new stream (e.g., for join) as new data arrive. Continuous query processing (CQP) on data streams requires advanced approaches which should reconsider most of the basics of queries on stored data [15]: not only transient, but also persistent (continuous) queries need to be processed; query answers are necessarily approximate due to the unboundedness of the stream leading to window joins to limit scope and to synopsis structures to approximate aggregates; because of the hardware constraints of mobile devices or/and the massive amounts of data that need to be processed, the size of the main memory is limited with respect to the data to be processed; data delivery is unpredictable.

Data streams can also be generated during query processing, as it happens in *on-line aggregation* in data warehousing contexts [11]. In this case, aggregation queries are employed to create a statistical result for decision making and represent one of the most expensive queries in database systems. In order to reduce the processing cost, and instead of changing the computing models, ApQP techniques, like online aggregation, can be used. The idea of online aggregation is to continuously draw samples from the database; the samples are then streamed to the query engine for processing the query and the query results are refined as more samples are retrieved. The processing strategy of online aggregation is similar to the one used in continuous queries on data streams. Continuous samples retrieved from the database can be considered as a stream and the query is evaluated against the stream. At a certain point of the computation, if the user is satisfied with the current results, she can terminate the processing to save the cost. If the user does not specify her preference, the query will run to the end and the precise results are returned.

Part III consists of two chapters that cover complementary issues concerning CQP.

Chapter 7 deals with the design and the implementation of join algorithms for data stream management systems, where memory is often limited with respect to data to be processed. A framework for progressive join processing, which can be used for various data models, is presented. Instantiations of the proposed framework

over different data models (relational, high-dimensional, spatial, and XML data) are also provided. Issues concerning the approximate processing of progressive join are finally discussed.

Online aggregation is the topic of Chapter 8. Besides introducing the basic principles of online aggregation, focusing on the sampling approach and the estimation model used to improve performance, the chapter also reviews some new applications built on top of it. Techniques for online aggregations in presence of multiple relations, multiple queries, distributed environments, and MapReduce architectures are discussed in details. Challenges of online aggregation and some future directions are also pointed out.

1.5 Adaptive Query Processing

The higher and higher resource sharing and the increasing interactivity in new processing environments make even those data properties that are traditionally conceived as static (such as relation cardinality and number of distinct values for an attribute) difficult to be known a priori and to be estimated. As a consequence, the traditional plan-first execute-next query processing model, according to which a query is processed following an execution plan selected on the basis of data statistics and a query optimization strategy, and then executed according to this plan with little or no run-time decision making, has begun to show its weaknesses. The need thus emerged to adapt the processing to dynamic conditions, revising the chosen execution plan on the fly, giving up the a priori selection of a single execution strategy, fixed before processing starts. In this case, query processing is called *adaptive* [6].

Adaptive query processing (AdQP) techniques have been proposed for various processing contexts, for both precise and approximate query execution. The main motivation for AdQP in local query processing is correcting optimizer mistakes, mainly due to unavailability of statistics as well as out of date statistics about attribute correlations and skewed attribute distributions. Parametric queries are another motivations for AdQP in this context. Similar issues arise in data management systems that support queries over autonomous remote data sources, in order to cope with query executions involving one or more sources for which no statistics are available. Adaptivity in distributed environments is also relevant in order to maximize CPU utilization, given the potentially changing rate at which data are received from the distributed data sources. Based on the characteristics of data streams, query operators and plans are necessarily adaptive: reacting to changes in input characteristics and system conditions is a major requirement for long-running query processing over data streams. For instance, stream arrival may be bursty, unpredictably alternating periods of slow arrival and periods of very fast arrival. The system conditions as well, e.g., the memory available to a single continuous query, may vary significantly over the query running time. In a situation where no input statistics

are known initially and input characteristics as well as system conditions vary over time, all the relevant statistics are estimated during execution.

Part IV consists of two chapters presenting problems, existing solutions, and trends in AdQP.

Chapter 9 surveys AdQP techniques for distributed environments. To this aim, a common framework is adopted for the classification of existing approaches. The framework decomposes the adaptivity loop into the monitoring, analysis, planning and actuation (or execution) phases. Differences between the main distributed AdQP techniques developed so far and their centralized counterparts are also discussed in details.

New trends in AdQP are considered in Chapter 10. In particular, after classifying adaptive and approximate techniques with respect to various parameters, including their goal (either Quality of Service (QoS)-oriented or Quality of Data (QoD)-oriented), the chapter shows that techniques applying QoD-oriented approximation in a QoD-oriented adaptive way, though demonstrated potentially useful on some examples, are still largely neglected. Some hints concerning how existing adaptive techniques can be extended to the new identified scenarios are also provided.

1.6 Queries over Heterogeneous Data Sources

The increasing number of independent data sources that are available for remote access by applications makes the problem of reconciling semantic heterogeneity among such sources a must. This problem has been traditionally referred to as *data integration problem* [2, 10] and has been the focus of attention for more than fifteen years, for both industry and academia.

Initial proposals for data integration approaches rely on a mediator-wrapper architecture by which a global schema is designed and the semantic integration is performed before query execution. Queries are then specified upon such global schema and specific wrappers are encharged of translating them upon the local source schemas [8]. Such kind of solutions works well assuming that the schema of each local source is known a-priori and static and that executable expressions can be derived in order to translate concepts appearing in one schema (the global one) into concepts appearing into another schema (a local one).

Unfortunately, such assumptions are not satisfied by data managed in new processing environments, where data sources to be integrated are selected and combined on-demand (as in mash-up applications [14]). As a consequence, the idea of *pay-as-you-go data integration* has emerged as an alternative approach to mediation-wrapper architectures. The basic idea is that in such new environments it is usually better to produce some tentative results in a short time than to have all precise results but waiting long or even to have nothing at all. This result can be generated in two steps through an adaptive process: first, a partial result is automatically generated, then it is continuously improved based on user feedback. Thus, pay-as-you-go

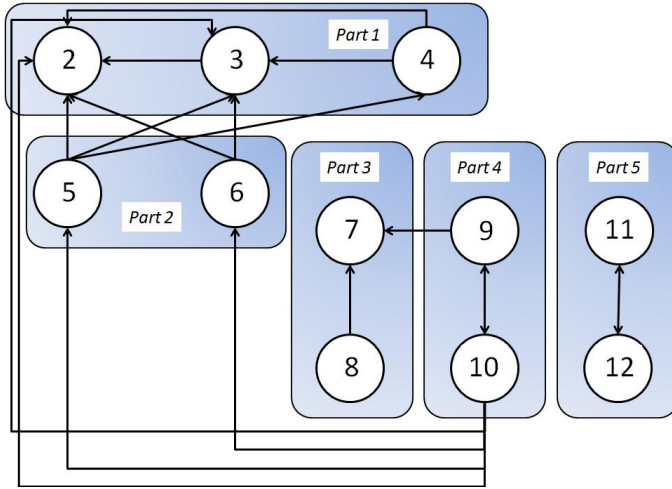


Fig. 1.2 Relationships between the chapters presented in the book.

integration approaches introduce a sort of approximation and adaptation in data integration architectures and, as a consequence, in queries executed upon them. *Dataspaces* [7, 9] represent an abstraction in data management upon which pay-as-you-go integration approaches have been proposed.

Part IV consists of two chapters, dealing with distinct approaches concerning integration and queries of HDS.

Chapter 11 deals with the problem of querying conflicting data spread over multiple web sources. First, a traditional solution to this problem is presented and limitations discussed; then, a more advanced approach to this problem is provided. Both approaches are illustrated in using genomic data sources accessible through the Web.

Dataspace management systems (DSMSs) are the topic of Chapter 12, where a comprehensive model of DSMS functionalities using an algebraic style is presented. To this aim, a dataspace life cycle is first provided. Then, the relationships between dataspace and automatic-mapping generation, based on model management techniques, are investigated. Relevant model-management capabilities are then formulated in an algebraic structure and the core functionality of a DSMS are characterized as a many-sorted algebra. As a consequence, the core tasks in the dataspace life cycle are represented as algebraic programs.

1.7 Conclusion and Discussion

In this chapter, we shortly discussed some of the most relevant issues arising in advanced query processing and we related them to the contributions presented in this

book. As pointed out by Figure 1.1 the considered issues have been rarely considered in isolation. To increase book readability, Figure 1.2 describes the relationships existing between the chapters presented in this book. An arrow from Chapter i to Chapter j means that Chapter i refers some of the issues discussed in Chapter j .

Of course, several additional topics related to advanced query processing exist which are not surveyed by this book. Among them, we recall: additional types of queries and related processing (e.g., keyword-based queries [4]); query processing for specific advanced data models and architectures (e.g., graph-based data models [5] and, more generally, noSQL databases [18]). Even if specific issues related to this topics are not addressed by this book, the main reference problems (preferences, approximation, adaptivity, data integration) are in common with those here discussed. We therefore hope that the presented contributions will also be useful to the readers in the investigation of further advanced query processing topics.

References

1. Agrawal, S., Chaudhuri, S., Das, G., Gionis, A.: Automated Ranking of Database Query Results. In: CIDR (2003)
2. Batini, C., Lenzerini, M., Navathe, S.B.: A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Comput. Surv.* 18(4), 323–364 (1986)
3. Börzsönyi, S., Kossmann, D., Stocker, K.: The Skyline Operator. In: ICDE, pp. 421–430 (2001)
4. Chaudhuri, S., Das, G.: Keyword Querying and Ranking in Databases. *PVLDB* 2(2), 1658–1659 (2009)
5. Cheng, J., Ke, Y., Ng, W.: Efficient Query Processing on Graph Databases. *ACM Trans. Database Syst.* 34(1), 1–48 (2009)
6. Deshpande, A., Ives, Z.G., Raman, V.: Adaptive Query Processing. *Foundations and Trends in Databases* 1(1), 1–140 (2007)
7. Franklin, M.J., Halevy, A.Y., Maier, D.: From Databases to Dataspaces: A New Abstraction for Information Management. *SIGMOD Record* 34(4), 27–33 (2005)
8. Halevy, A.Y.: Answering Queries Using Views: A Survey. *The VLDB Journal* 10(4), 270–294 (2001)
9. Halevy, A.Y., Franklin, M.J., Maier, D.: Principles of Dataspace Systems. In: PODS, pp. 1–9 (2006)
10. Halevy, A.Y., Rajaraman, A., Ordille, J.J.: Data Integration: The Teenage Years. In: VLDB, pp. 9–16 (2006)
11. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online Aggregation. In: SIGMOD Conference, pp. 171–182. ACM Press (1997)
12. Ilyas, I.F., Beskales, G., Soliman, M.A.: A Survey of Top- k Query Processing Techniques in Relational Database Systems. *ACM Comput. Surv.* 40(4), 11:1–11:58 (2008)
13. Koutrika, G., Ioannidis, Y.: Personalizing Queries based on Networks of Composite Preferences. *ACM Trans. Database Syst.* 35(2), 1–50 (2010)
14. Lorenzo, G.D., Hacid, H., Paik, H.Y., Benatallah, B.: Data Integration in Mashups. *SIGMOD Record* 38(1), 59–66 (2009)
15. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G.S., Olston, C., Rosenstein, J., Varma, R.: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In: CIDR (2003)

16. Rigaux, P., Scholl, M., Voisard, A.: Spatial Databases - With Applications to GIS. Elsevier (2002)
17. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access Path Selection in a Relational Database Management System. In: SIGMOD Conference, pp. 23–34 (1979)
18. Stonebraker, M.: SQL Databases vs. NoSQL Databases. Commun. ACM 53(4), 10–11 (2010)
19. W3C: XML Path Language (XPath) 2.0 (2007),
<http://www.w3.org/TR/xpath20/>
20. W3C: XQuery 1.0: An XML Query Language (2007),
<http://www.w3.org/TR/xquery/>

Chapter 2

On Skyline Queries and How to Choose from Pareto Sets

Christoph Lofi and Wolf-Tilo Balke

Abstract. Skyline queries are well known for their intuitive query formalization and easy to understand semantics when selecting the most interesting database objects in a personalized fashion. They naturally fill the gap between set-based SQL queries and rank-aware database retrieval and thus have emerged in the last few years as a popular tool for personalized retrieval in the database research community. Unfortunately, the Skyline paradigm also exhibits some significant drawbacks. Most prevalent among those problems is the so called “curse of dimensionality” which often leads to unmanageable result set sizes. This flood of query results, usually containing a significant portion of the original database, in turn severely hampers the paradigm’s applicability in real-life systems. In this chapter, we will provide a survey of techniques to remedy this problem by choosing the most interesting objects from the multitude of skyline objects in order to obtain truly manageable and personalized query results.

2.1 Introduction

The ever growing amount of available information is one of the major problems of today’s information systems. Besides solving the resulting performance issues, it is imperative to provide personalized and tailored access to the vast amount of information available in information and data systems in order to avoid flooding the user with unmanageably large query results.

Christoph Lofi
Technische Universität Braunschweig, Germany
e-mail: lofi@ifis.cs.tu-bs.de

Wolf-Tilo Balke
Technische Universität Braunschweig, Germany
e-mail: balke@ifis.cs.tu-bs.de

As a possible remedy to this problem, Skyline queries [1] have been proposed, filling the gap between set-based SQL queries and rank-aware database retrieval [2]. Due to the paradigm elegance and simplicity, it has stirred a lot of interest within the database community in recent years. Skyline queries rely on the notion of *Pareto dominance*, i.e., given the choice between two objects, with one object being better with respect to at least one attribute but at least equal with respect to all other attributes, users will always prefer the first object over the second one (the first object is said to *dominate* the second one). This simple concept can be used to implement an intuitive personalized data filter as dominated objects can be safely excluded from the data collection, resulting in the *Skyline set* of the query. The semantic justification of this filter is easy to see using an example: if two car dealers in the neighborhood offer the same model (with same warranties, etc.) at different prices, why should one want to consider the more expensive car?

In order to compute the Skyline set in a personalized fashion, the user needs only to provide so-called *ceteris paribus* (“all other being equal”) preferences on each individual attribute (e.g., “lower prices are better than higher prices given that all other attributes are equal”). Although, many works on skyline queries only consider numerical domains and preferences [1,3,4], skylining can generally also be extended to qualitative categorical preferences (e.g., on colors, “given two cars with free color choice, a black car would be better than a red car”) which are usually modeled as partial or weak orders [5,6]. Furthermore, many of these preferences don’t require any user input during elicitation as they can be deduced from common content in the collection of user profiles (e.g., preferences on price; no reasonable user would prefer the same object for a higher price).

This focus on individual attribute domains and the complete fairness of the Pareto paradigm are the major advantages of skyline queries: they are easy to specify and the algorithm will only remove definitely suboptimal objects. However, these characteristics also directly lead to the paradigm’s major shortcomings: Skyline queries completely lack the ability to relate attribute domains to each other and thus prevent compensation, weighting or ranking *between* attribute domains. This often results in most objects being incomparable to each other and thus generally causes skyline sets to be rather large, especially in the quite common case of anti-correlated attribute dimensions. This effect is usually referred to as “curse of dimensionality”. It has been shown (under certain assumptions on e.g. the data distribution) that the skyline size grows roughly exponential with the number of query attributes [7,8]. However, there is still no reliable and accurate algorithm for predicting skyline sizes given arbitrary database instances and user preferences. Experimentally, it has been validated that already for only 5 to 10 attributes, skylines can easily contain 30% or more of the entire database instance [1,9,10] which is a size clearly unmanageable for most users, rendering the skyline paradigm inapplicable for many real-world problems.

Thus, reducing the size of result sets by choosing the most interesting or most relevant objects from the skyline is a major and prominent problem. However, “interestingness” is usually individual perception and is specific for each user and is thus hard to formalize. Nevertheless, for rendering the skyline paradigm useful for common real world scenarios, such techniques are mandatorily required. Accordingly, an impressive number of approaches have been developed in the recent years

introducing various heuristics for capturing the semantics of “interesting” in order to choose meaningful and manageable subsets from skylines in an efficient manner.

Generally speaking there are four major approaches to address the problem:

- *Relaxation of Pareto Semantics* use weaker variants of the Pareto semantics which less likely lead to incomparability between database objects. These approaches include for example Weak Pareto Dominance or k-Dominant Skylines and are discussed in Section 2.3.
- *Summarization approaches* are presented in Section 2.4 and aim at returning a representative subset which still maintains the diversity and flavor of the original skyline set. Often such approaches are intended to enable the user to grasp a quick overview of the whole skyline set. Examples are Statistical Sampling Skylines and Approximately Dominating Representatives.
- *Weighting approaches* try to induce a ranking on the Pareto incomparable skyline items based on some structural or statistical properties of the data set. Usually, they numerically quantify the “interestingness” of a skyline object explicitly and return the k-most interesting objects. These approaches are showcased in Section 2.5 and are often based on extensive subspace skyline computation. They include for example Top-K Frequent Skyline, Skyrank, or Personalized Top-k Retrieval.
- *Cooperative approaches*, presented in Section 2.6, interactively try to elicit more information from users to refine the preferences in order to focus the skyline sets in a personalized fashion. While abstaining from using heuristics for selecting the skyline objects, they impose an additional interaction overhead on the user. These approaches include for example Trade-Off Skylines.

After introducing the basic formalities necessary for modeling preferences and computing skylines, we will present selected techniques for each of these major approaches.

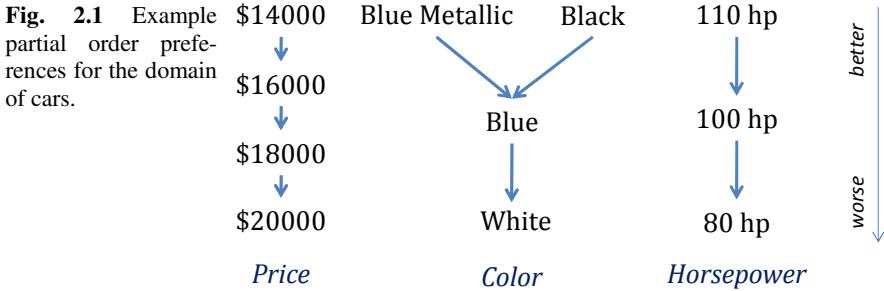
2.2 Formalization of Skyline Sets Following the Pareto Semantics

Before we survey the field of skyline selection algorithms let us formally introduce the skyline paradigm and the underlying Pareto semantics. Skyline sets can be defined for every database relation $R \subseteq D_1 \times \dots \times D_n$ over n attributes. Assessing a preference for the values in domains D_i of each attribute, users can provide a set of up to n complex attribute preferences to personalize the skyline retrieval process:

- A preference P_i on an attribute A_i with domain D_i is a *strict partial order* over D_i . If some attribute value $a \in D_i$ is preferred to some other value $b \in D_i$, then $(a, b) \in P_i$. This is often written as $a >_i b$ (read “ a dominates b wrt. to P_i ”).
- Analogously, an equivalence Q_i on some attribute is an *equivalence relation* on D_i compatible with P_i (i.e., no tuple in Q_i may contradict a tuple in the transitive closure of P_i). If two attribute values $a, b \in D_i$ are equivalent, i.e., $(a, b) \in Q_i$, we write $a \approx_i b$.

- Finally, if an attribute value $a \in D_i$ is either preferred over, or equivalent to another value $b \in D_i$, we write $a \succeq_i b$.

As an example, assume some preferences for buying a car considering the attributes price, color, horsepower, and air conditioning as illustrated in Figure 2.1.



Skyline sets as introduced in [1] are defined using the Pareto semantics from the field of economy [11]: some object o_1 dominates an object o_2 , if and only if o_1 is preferred over o_2 with respect to any attribute and o_1 is preferred over or equivalent to o_2 with respect to all other attributes. Formally the dominance relationship is denoted as $o_1 > o_2$ and can be expressed as given in Definition 2.1.

Definition 2.1 (Dominance Relationships following Pareto Semantics).

$o_1 > o_2 \Leftrightarrow \exists i \in \{1, \dots, n\}: o_{1,i} >_i o_{2,i} \wedge \forall i \in \{1, \dots, n\}: o_{1,i} \succeq_i o_{2,i}$ where $o_{j,i}$ denotes the i -th component of the database tuple o_j .

The skyline set (or Pareto skyline) can then be defined as the set of all *non-dominated* objects of the database instance R with respect to all preferences following the Pareto semantics:

Definition 2.2 (Skyline Set).

$$Sky := \{o_1 \in R \mid \neg \exists o_2 : o_2 > o_1\}$$

For actually computing a skyline set, there are multiple algorithms available which can be classified into Block Nested Loop algorithms, Divide-and-Conquer algorithms, and Multi-Scan Algorithms. Basically, each object has to be compared to each other one and tested for dominance. However, advanced algorithms try to avoid testing every object pair by employing optimizations and advanced techniques to eliminate as many objects as possible early in the computation process. Efficient skyline algorithms thus require only a fraction of the number of object dominance tests than less sophisticated algorithms.

Block-Nested-Loop (BNL) algorithms are probably the most popular algorithm class and were developed quite early [1]. However, also many state-of-the-art algorithms use the BNL approach [12-16]. These algorithms scan linearly over the

input database and maintain a list containing the current intermediate skyline (called windows or block). Each newly scanned object is compared to the objects in the windows, eliminating dominated objects or being eliminated and discarded. If the object is not dominated by any object in the window, it is also added to the window. More sophisticated version of this algorithm maintain the basic design principles of using a single scan and maintaining a window, but employ additional techniques like presorting or indexing to increase the overall performance. Due to the single-scan nature, these algorithms are especially suited to be used in database systems which are often optimized for linear access.

The second popular class of skyline algorithms are Divide-and-Conquer approaches which recursively split the input data and then joins the partial skylines. Although these algorithms have excellent theoretical properties [1,7], there is no efficient implementation of this recursive process [12]. Thus, this algorithm class is very popular from a theoretical point of view but rarely used in actual software systems.

The third class of skyline algorithms is based on multiple scans of the database instance and includes algorithms like Best or sskyline [17-19]. They can especially provide highly efficient cache-conscious implementations. These algorithms may eliminate objects especially early, but require scanning and modifying the input database numerous times. Thus these algorithms are mainly used when the whole input relation fits into main memory.

From an *order-theoretical point of view*, in skyline computations all attribute preferences have to be aggregated, thus forming the *full product order* P . This product order materializes all dominance relationships between all possible database objects. The Skyline then consists of all those objects *existing in* R which are not dominated by other *existing objects* with respect to P . The formal notion of the full product order is given by Definition 2.3. However, for complexity reasons skyline algorithms obviously cannot materialize the full product order. Nevertheless in later sections, we will encounter approaches relying on the materialization of at least parts of the full product order.

Definition 2.3 (Full Product Order P).

The full product order is given by $P \subseteq (D_1 \times \dots \times D_n) \times (D_1 \times \dots \times D_n)$, where for any $(o_1, o_2) \in P$ holds $o_1 > o_2$. The semantics of $>$ are given by the Pareto dominance in Definition 2.1.

As an example for Pareto skylines, consider Figure 2.2: On the left-hand side of the figure, two partial-order attribute preferences P_1 and P_2 are given. The resulting object order P of seven example database objects is shown in the top of the figure. Only two objects are dominated using Pareto semantics, thus five objects form the skyline. In particular, note that object $(i, 1)$ is in the skyline as the attribute value i is isolated in the preference P_1 , i.e., no database object may ever dominate $(i, 1)$.

Please note that partial order preferences reflect an intuitive understanding of preferences given by simple statements for each attribute like “I like A better than B”. But when relying on partial orders the additional possibilities for objects

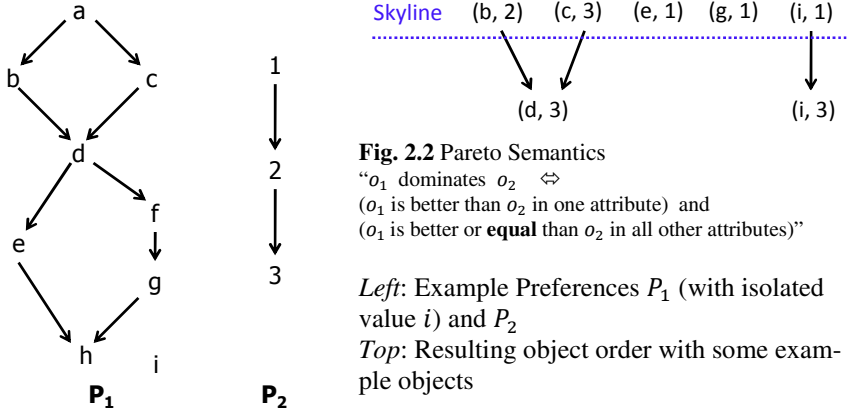


Fig. 2.2 Pareto Semantics

“ o_1 dominates $o_2 \Leftrightarrow$

(o_1 is better than o_2 in one attribute) and

(o_1 is better or **equal** than o_2 in all other attributes)”

Left: Example Preferences P_1 (with isolated value i) and P_2

Top: Resulting object order with some example objects

being incomparable may introduce efficiency issues for the algorithm design (especially by preventing effective, yet simple pruning conditions). Hence the first introduction of Skylines in [1] only dealt with attribute scorings (i.e., weak order preferences). While this allowed for very efficient query evaluation, the preferences’ expressiveness was rather limited [20]. But this drawback was quickly remedied by [21] and [22], which both helped to popularize the use of partial order preferences.

There are multiple reasons for objects being incomparable (e.g., no object can dominate the other one): a) two objects are incomparable if they have antagonistic attribute values, e.g., when considering two cars, one with (75 HP, 5 Liter / 100km) and one with (120 HP, 9 Liter / 100km), these two cars would be incomparable when using the default preferences “more HP is better” and “lower fuel consumption is better” as none of the two objects is clearly better than the other. b) two objects are incomparable if there is no relationship defined between the respective attribute values (often referred to as “missing information”). For example, considering the preferences in Figure 2.2 above, the objects $(e, 1)$ and $(f, 3)$ are incomparable because there is no information on whether e is preferred over f or vice versa. This effect is especially severe for any object sporting an isolated attribute value like e.g. $(i, 2)$. Incomparability due to missing information is a problem exclusive to partial order preferences and is non-existent for total or weak orders. c) two objects are incomparable due to the user being indifferent with respect to certain attribute values. This happens commonly for weak orders where there are equivalence classes of attribute values which are considered equally preferred. But also partial order preferences commonly allow for explicitly modeling equivalences. As an example, consider a partial user preference with an equivalence statements “Red is as desirable as yellow”. Then two completely similar objects with one being red and the other being yellow would be incomparable, and both could be potential skyline objects.

2.3 Relaxing the Pareto Semantics

As we already argued, the major problem of skyline queries are the often unmanageable result sets possibly dumping thousands of items on the user for manual inspection. Considering the definition of Pareto semantics, it is obvious that the manageability problems of skylines are heavily aggravated by incomparable attribute values. As soon as two database items are incomparable with respect to even a single attribute, the entire objects are incomparable and may both end up in the skyline. One could say that the Pareto semantics generally is ‘too fair’. The obvious solution to restrict skyline sizes is to move from full-fledged partial order preferences to total attribute orders (or at least weak orders), but this in turn might cause problems in the preference elicitation process (more user interactions, sometimes even the introduction of cyclic preferences). The question is whether it is possible to keep partial order preference semantics while adapting the Pareto semantics’ fairness to reduce skyline sizes. In this section, we will present skyline approaches based on partial orders which rely on a weaker definition of dominance relationships than given in Definition 2.1 in order to obtain more manageable result sets: weaker dominance conditions mean more object dominance relationships and thus smaller skyline sets. From an algorithmic point of view, these algorithms stay similar to normal Skyline algorithms and just use a different notion of dominance. However, in some cases (like e.g. for weak dominance), the weaker dominance conditions may allow for additional pruning heuristics which may increase the algorithms efficiency.

A first straightforward approach is to explicitly derive weak orders from given partial orders, leading to so-called *level order skylines*. Here counting from the most preferred values down to the least preferred ones, all values are assigned a level. Now any attribute value can be considered dominated by all values on higher levels

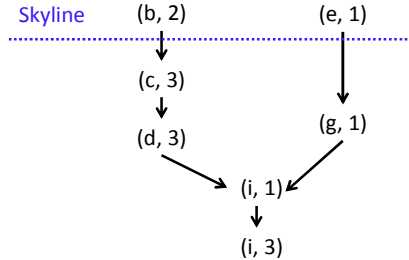
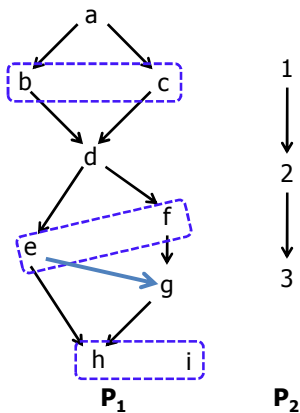


Fig. 2.3 Level-Order Semantics
 “ o_1 dominates o_2 : \Leftrightarrow
 (o_1 is better than o_2 in one dimension
 and (o_1 has a higher or the same **level** as o_2 in all other dimensions)”

Left: Example Preferences P_1 and P_2 with levels order equivalences

Top: Resulting object order with example objects

(see Figure 2.3). Especially, isolated values for which the user did not provide any preference are considered to be part of the lowest level (and are thus easily dominated).

This concept can be extended into *weak* attribute dominance by further loosening up the dominance semantics. The Pareto semantics and its previously presented variants require an object o_1 in order to dominate an object o_2 to be better in one dimension and at least equal in all others. In case of an object being incomparable with respect to one of the dimensions, no dominance relationship can be established.

However, the basic intuition of skylining is the result should contain all “best” objects – “best” could be interpreted as “there are no better objects”. This interpretation can be formalized into the weak attribute dominance as illustrated in Figure 2.4 and can be phrased as “ o_1 dominates o_2 if o_1 is better than o_2 in one dimension and there is no dimension such that o_2 is better than o_1 ”. In general, this dominance criterion leads to significantly reduced result sets (called weak Pareto skylines, and defined by imposing the Pareto semantics on the weak attribute dominance relationships). Weak Pareto skylines can be quite efficiently computed, see [23]. However, the size reductions are usually quite significant, and often even desired skyline objects are removed. When considering the example in Figure 2.4, it can be observed that the isolated object $(i, 1)$ now dominates the object $(b, 2)$ which shows an overall solid performance, and thus might have been a good choice for many users which is not available anymore.

A less aggressive dominance criterion is provided by the *substitute value* (SV) semantics in [24] which consider Pareto incomparable values within one attribute preference as being equal if they share the same parents and children within the partial preference order. Naturally, the resulting SV skylines will be larger than a corresponding weak skyline.

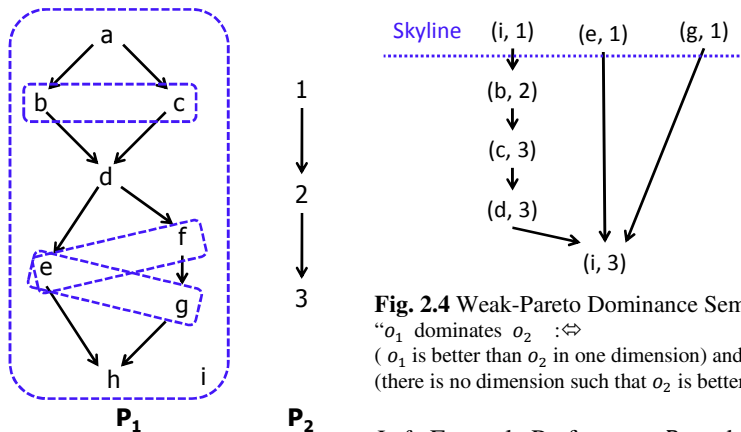


Fig. 2.4 Weak-Pareto Dominance Semantics
 “ o_1 dominates o_2 \Leftrightarrow
 (o_1 is better than o_2 in one dimension) and
 (there is no dimension such that o_2 is better than o_1)”

Left: Example Preferences P_1 and P_2 with weak-Pareto equivalences

Top: Resulting object order and restricted skyline

In [25], the concept of k -dominant skylines is introduced, based on the measure of k -dominance. An object o_1 is said to k -dominate another object o_2 if there are at least $k \leq d$ dimensions in which o_1 is better than or equal to o_2 and o_1 is strictly better in at least one of these k dimensions. Using this definition, a Pareto skyline is a k -dominant skyline with $k = d$. K -dominant skylines with $k < d$ are motivated by the observation that with the growing number of dimensions, it becomes highly unlikely that an object is equal or worse with respect to all dimensions than another one as most objects have at least some strong attribute values in a high dimensional space. Thus, rarely any dominance relationships can be established. K -dominant skylines allow objects exhibiting only few good attribute values to be dominated by objects showing many good attribute values. By decreasing the user provided k , the resulting k -skylines are decreased in size. K -Skylines with smaller k are subsets of those with higher a value for k , e.g., given a fixed dataset, the 2-skyline is a subset of the 3-skyline, etc. All k -skyline are a subset of the original skyline. Computing k -dominant skylines requires specialized algorithms which are also provided in the work.

2.4 Summarizing the Skyline

In this section, approaches are presented which aim at finding a subset of objects which serves optimally as a summarization of the full skyline. The main idea behind these approaches is to return just a summarizing set of objects which still maintains the diversity and characteristics of the original skyline, but exhibits a much more manageable size, i.e., sacrificing the completeness of skylines for the sake of manageability. The focus within these approaches is to enable the user to grasp a quick overview of the nature and contents of the skyline result set such that she is easily able to further refine her preferences and / or is directly able to perform subsequent queries for narrowing down the results even further (e.g., appending a top- k query which ranks the skyline result, or provide some SQL constraints to remove unwanted data points).

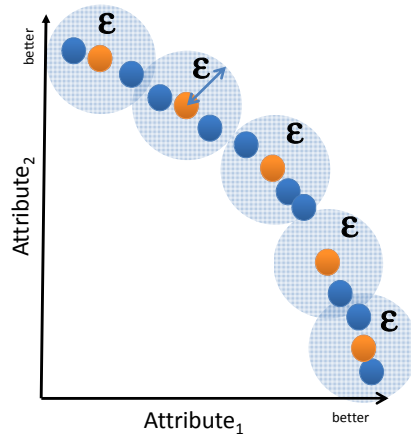
We will focus on two approaches: a) approximately dominating representatives [26] which returns a subset minimally covering all skyline objects within ϵ -balls and b) statistical sampling approaches [10] with subsequent top- k ranking. Both approaches try to maintain the diversity of the original skyline and do not suggest a ranking of skyline objects.

2.4.1 Approximately Dominating Representatives

Computing *approximately dominating representatives* (ADR) [26] is a subsequent refinement technique of skyline queries which aims at covering the full skyline by an optimal sample in terms of size and accuracy. The ADR query returns a set of k (whereas k is a user specified value) objects $adr_\epsilon := \{a_1, \dots, a_k\}$ with the following property: based on a user specified value ϵ , for any other object $a \notin adr_\epsilon$ holds that there is an $i \leq k$ such that the vector $a_i \cdot (1 + \epsilon)$ (i.e., a_i boosted by ϵ

in all dimensions) dominates a . In other words, ADR tries to find a minimal number of skyline objects such that, if they are assumed to be the center of a sphere with radius ϵ in the data space, all other skyline objects (which are not the center of such a sphere) are within one of the spheres around the selected skyline objects (see Figure 2.5).

Fig. 2.5 Approximately Dominating Representatives with ϵ -spheres. Only the center objects of each ϵ -sphere is returned.



ADR is designed as a subsequent step after the actual skyline computation, thus no performance advantages can be gained as still the full skyline needs to be computed. Furthermore it has been shown that although finding the smallest possible ADR (assuming the skyline is given) is in linear time complexity in the number of skyline elements for two attribute dimensions, the problem is unfortunately NP hard for more than two dimensions. Accordingly, approximation algorithms have been developed which run in polynomial time, but sacrifice some of the accuracy of the cover.

The resulting cover will contain some objects from the full spectrum of the skyline, disregarding any additional semantic or structural properties. For example, when considering a simple two dimensional skyline on cars with the attributes top speed and price, it will contain all flavors of different cars with respect to those two attributes: very fast cars which are very expensive, many in-between variations of less expensive but slower cars, down to very slow and very cheap cars.

2.4.2 Statistical Sampling Skylines

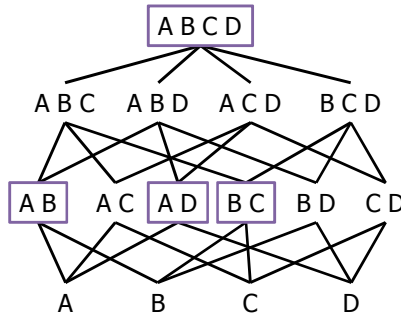
Sampling skylines [10] follow a similar base idea as they are designed to return a representative subset of the original skyline. However, the *sampling skyline* approach is specifically designed for cooperative retrieval scenarios and is intended to precede a later query-by-example user interaction. Compared to approximately dominating representative skylines, the focus is more on computational performance and less on the accuracy and representativeness of the returned sample.

The intended user workflow is as follows: a) quickly generate a sample set from the database using the sampling skyline algorithm (this set is intended to be representative for the most interesting objects of the database according to some provided attribute preferences); b) elicit feedback from the user on which objects from the sample she prefers; c) discover common characteristics of the chosen objects algorithm, e.g., in order to derive a top- k utility function; d) return a ranked list of best database objects to the user.

The main goal of the sampling algorithm is that the resulting sample can be computed fast (this especially means that it can be computed significantly faster than the full skyline), and that the result is representative for the full skyline in terms of the object diversity. Also, the size of the sample should be small and manageable.

The algorithm is based on the idea that for sampling a skyline with n query attributes, q subspace skylines on randomly selected subspaces with a dimensionality of $m < n$ are computed and then summarized (thus, this approach is a randomized variant of the subspace analysis-based approaches presented in the next section). Each of these q subspace skylines represents a “topic of interest” and contains objects which are “optimal” considering the respective focus of the chosen subspace. Furthermore, these subspace skylines can be computed extremely fast compared to computing the full skyline. By generating a duplicate-free union of the cleaned subspace skylines, a sample is obtained (see Figure 2.6).

Fig. 2.6 Statistical Sampling Skylines. A skyline query on four attributes (A, B, C, D) is sampled by a union of the randomly selected subspace skylines with $m = 2$ dimensions (A, B), (A, D), and (B, C).



This sample can be shown to be statistically representative for certain values of m and q , leading to a sufficient number of different topics of interest to be covered, e.g., considering a skyline of a car database, the sample could contain some fuel efficient cars, some cheap cars, some fast cars, and some luxurious cars. Furthermore, the sample contains only real skyline objects and can be computed significantly faster than the full skyline.

2.5 Weighting Characteristics of Skyline Points

Approaches presented in this section also aim at generating a sample of the skyline. However, in contrast to the approaches presented in the previous section which try to cover the full diversity of different skyline objects, the following

section presents approaches which select and also rank a subset of the skyline due to some explicitly modeled measure of “interestingness”. Especially, in contrast to summarizing techniques, objects showing some rare and extreme values are often not considered as being interesting by these approaches. The Pareto skyline operator treats all skyline objects as being equal, i.e., it does not impose any ranking on the result set. However, the following approaches claim that there are more important and less important skyline objects, and that “importance” can be captured by properties like e.g. the data distribution, the structure of the subspace skylines, or other statistical means.

The first approaches presented in this section will focus on the analysis of subspace skylines, i.e., objects are more interesting depending on in which and in how many subspace skylines they appear. The latter approaches will try to capture semantic importance of skyline objects by relying on the number of objects a given skyline object dominates.

2.5.1 *Skycubes and Subspace Analysis*

Focusing on computing and analyzing subspace skylines is a very popular approach to derive rankings of Skyline objects. However, computing a larger number of subspace Skylines is prohibitively expensive for online algorithms. Thus, in order to enable the extended use of subspace analysis, *skycubes* [27] have been developed. Skycubes are similar to datacubes used in data warehousing in that respect that they contain all precomputed non-empty skylines of all (possibly up to $2^d - 1$) subspaces of a given dataset. By precomputing all these skylines, drill-down analysis or subspace skyline membership queries can be answered quickly and efficiently. Furthermore, when computing all subspace skylines for a skycube, specialized algorithms can be used which rely on different computation sharing techniques. Thus, computing a whole Skycube is more significantly more efficient than computing all subspace skylines individually.

Mainly, two methods have been proposed to compute all skylines for all subspaces, both relying on traversing the lattice of subspaces (see, e.g., Figure 2.7) either in a top-down or bottom-up manner. In the bottom-up approach, the skylines in a subspace are partly derived by merging the skylines from its child subspaces at the lower level. In the top-down approach, recursively enumerate all subspaces and compute their skylines from the top to bottom level. During this computation, lower subspace skylines may reuse results from higher subspace skylines, thus significantly conserving computation time. This turns out to be much more efficient than the bottom-up approach. Ultimately, these algorithms allow for computing a skycube up to two magnitudes faster than computing all subspace skylines individually.

With having all subspace skylines readily available due to the materialization of a Skycube, the path is cleared for extensive subspace analysis. For example in [28], the semantics of subspace skylines are explored and researched. Especially, the concept of decisive subspaces and skyline groups as semantically interesting applications of subspace analyses is introduced. As this type of Subspace analysis

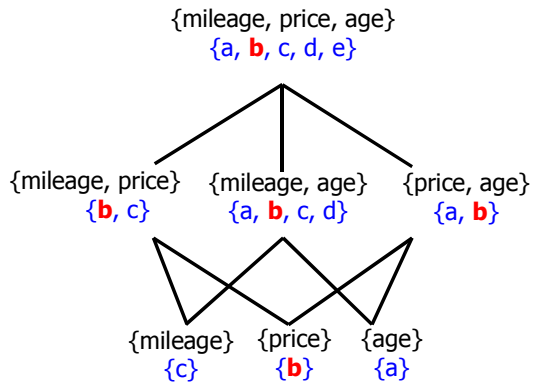
synergizes greatly with the skycube algorithm, even a combined research work unifying both approaches was later provided in [29].

A crucial concept in subspace analysis of skylines are skyline groups. Skylines in subspaces consist of projections of objects. For a projection of an object that is in the skyline of a subspace, the set of objects that share the same projection form a so called coincidence group. The maximal coincidence group for a subspace is called skyline group. Whether an object is in the skyline of the full space or of some subspaces is determined by the values of the object in some decisive subspaces, i.e., those smallest subspaces for which an object is part of a skyline group. The decisive subspaces and the values in those subspaces vary from object to object in the skyline. For a particular object, the values in its decisive subspaces justify why and in which subspaces the object is in the skyline and thus represent the deciding semantics for the membership of the object in the skyline. It has been suggested that Skyline groups can be used to summarize the full skyline by selecting some common representatives from the skyline groups of the skycube.

Fig. 2.7 Skycube Lattice of a car database on attributes *mileage*, *price*, *age* and skyline objects $\{a, b, c, d, e\}$

Top-1 Frequent Skyline: $\{b\}$

Top-3 Frequent Skyline: $\{b, a, c\}$



In [30], it is suggested that a metric called *skyline frequency* can be used to rank and select skyline objects by their interestingness. Skyline frequency counts for each skyline object the number of its occurrence in all possible non-empty subspace skylines, claiming that skyline objects with a higher subspace skyline frequency are more interesting to users than those with lower frequencies. This approach is extended by using skyline frequency to present the user with a ranked list of k most “interesting” skyline objects (so called *top- k frequent skyline*). As an example for this approach consider a skyline query on a used car database with the dimensions mileage, price, and age (see Figure 2.7). The most frequent skyline object in all subspaces is b , while a and c are similar frequent.

This approach can be seen as a fusion of top- k queries and skyline queries which retains the easy and intuitive query specification of skyline queries, but also allows for limiting the number of objects in a result set to the k most interesting objects which are additionally ranked. The ranking function relies purely on structural properties of the underlying subspace skylines, no additional user input is necessary.

Specialized algorithms are provided for computing the top-k frequent skyline, in an either exact, or due to the high computational complexity, approximate manner. Also, pre-computed skycubes can be used to further accelerate the computation of the top-k frequent skyline.

2.5.2 SKYRANK

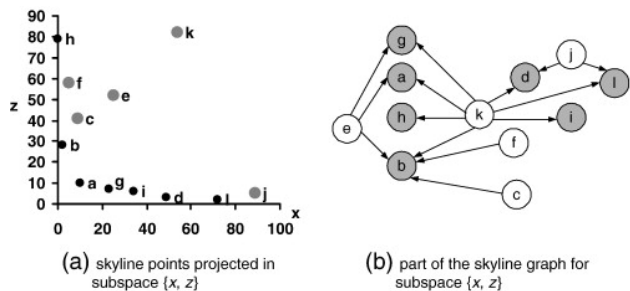
The SKYRANK algorithm [31] also relies on ranking skyline objects based on their subspace relationships. However, the measure of “interestingness” is not just based on the number of appearances of an object in any of the subspace skylines, but on the dominance relationships of the skyline objects in the respective subspaces. Following heuristic provides the foundation of SKYRANK:

- A skyline object is more interesting if it dominates many other important skyline points in different subspaces.
- The interestingness of a skyline object is propagated to all skyline object that dominate it in any subspace.

This means, the interestingness of an object in the full space skyline increases the more other skyline objects it dominates in any of the subspaces. Furthermore, this interestingness is amplified if those objects it dominates have been especially interesting themselves (i.e., were also dominating many original skyline objects in some subspaces).

To compute this recursive concept of interestingness, SKYRANK relies on the notion of a skyline graph. The skyline graph contains all skyline objects of the full data space as nodes. The edges are constructed using the skycube of a dataset. For every skyline object o of the full dimensional space (the nodes of the graph), each subspace of the skycube is tested if o is part of the corresponding subspace skyline. If it is, no edge is added. However, if o is not part of the subspace skyline, a directed edge is added from o to all objects which dominate o with respect to the current subspace. This concept is illustrated in Figure 2.8 for a part of the skyline graph of the full skyline graph just focusing on a two dimensional subspace $\{x, y\}$.

Fig. 2.8 Part of a Skyline graph the subspace $\{x, z\}$ with the respective subspace skyline. Darker nodes / objects are subspace skyline objects.



The idea for capturing the semantics of “propagating interestingness” is to use link-based ranking methods. For SKYRANK, the PageRank [32] algorithm was chosen which was originally designed for ranking web pages for web search engines. The algorithm models the behavior of a random web surfer and iteratively approximates the stationary probability of the surfer visiting a web page (represented by the PageRank score). The basic heuristic is that web pages with many in-links have more authority and are thus more important than those with few in-links, and also that a page may transfer some of its authority to another page by linking to it. The authors of SKYRANK claim that a skyline graph implies similar semantics than a web graph: by linking to an object o_1 , an objects o_2 transfers some of its authority to o_1 (i.e., o_2 is dominated by o_1 in some subspace, thus o_1 must be more important). Furthermore, objects with many in-links are more important than those with few in-links (they dominate more other objects).

Accordingly, SKYRANK presents algorithms for efficiently constructing skyline graphs and applies a link-analysis algorithm similar to PageRank in order to compute scores for each skyline object. These scores can be used to select a ranked list of the k most interesting skyline objects. Furthermore, the algorithm can be personalized by providing top- k -style weightings for all of the involved subspaces, thus allowing for an either user agnostic or personalized retrieval model.

2.5.3 k Most Representative Skyline Points

In contrast to the previous approaches which rely on the nature of subspace skylines, the following approach aims at summarizing a skyline by using the k most representative skyline points (RSP) [33]. For capturing the semantics of this skyline selection, RSP defines the concept of “representative” by the population, i.e., a skyline point is more representative the more objects it dominates. This representativeness metric can be used to rank skyline objects. Finally, the k -most representative problem is stated as finding the k skyline points (whereas k is given by the user) with the maximal number of dominated database objects.

This approach also incorporates ideas from top- k retrieval into skyline queries by returning the k most interesting objects in a ranked fashion. Again, the selection and ranking is based on purely structural properties of the skyline objects, not requiring additional user feedback.

The underlying problem of this operation is similar the set cover problem which also underlies the approximate dominating representative approach in Section 2.4.1. Thus, the runtime complexity is also similar: for more than three query dimensions, the problem is NP-hard, but can be approximated by greedy heuristics. In addition, the authors provide a more efficient randomized approach with a better scalability and establish theoretical accuracy guarantees.

Furthermore, this approach has been extended in [34] to retrieve the k representative skyline points defined as the set of k points that minimize the distance between a non-representative skyline point and its nearest representative.

2.5.4 Personalized Top- k Retrieval / Telescope

In [35], the authors propose a more personalized approach to selecting the k best ranked skyline objects. Instead of relying on user oblivious ranking functions like the previous works on k -dominance or k -representatives, the ranking of skyline objects is steered by additional user provided preferences for weighting individual attribute dimensions (but is still refraining from unintuitive quantitative ranking functions as used by traditional top- k querying). Thus, this approach bridges into the area of cooperative approaches presented in Section 2.6 which focus on additional user interactions for selecting skyline subsets, but which also avoid ranking of skyline objects.

In addition to the attribute preferences required to compute the skyline, the user provides preferences expressing a *precedence order* of the involved attribute dimensions. For example, assuming a used car database, a user can provide attribute preferences like e.g. higher fuel efficiency is better, lower prices are better than higher prices, and red is better than blue which is better than yellow. Additionally, she may state that color is more important to her than for example the price.

For actually returning a ranked skyline containing k objects, a data structure similar to a skycube lattice is traversed based on the precedence preferences. For each level, the subspace nodes are ordered with respect of the precedence preference with subspaces containing more important dimensions to the left. The traversal starts at the top node representing the skyline of the full query space and an empty result set. The actual navigation follows one of two access modes:

- Vertical to left-most child: If the current node contains too many skyline objects (i.e., more than k - size of current result set), the evaluation navigates to the child subspace with the highest precedence (i.e., the left most child).
- Horizontal to right sibling: If the current node contains not too many objects (i.e., less than k - size of the current result set), all skyline object of the current subspace are added to the result set and the evaluation continues with the next sibling to the right (the subspace with the next highest precedence).

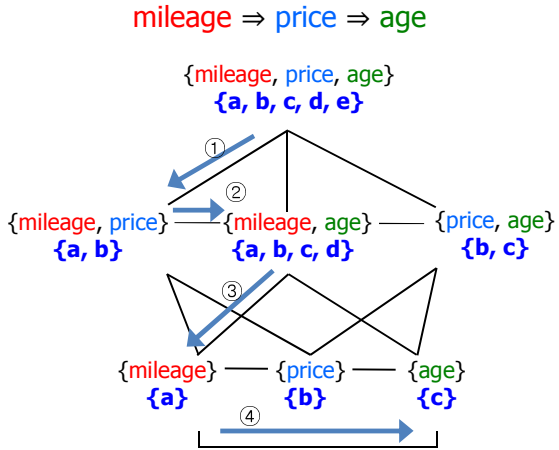
This navigation model is further illustrated in Figure 2.9 for a used car database. The provided precedence is “low mileage is more important than low price, which is more important than young age”, and $k = 3$ skyline objects are to be returned. In the first step, the algorithm starts at the root node of the full subspace which contains more than 3 objects. Thus, it navigates to the next lowest subspace with highest precedence $\{mileage, price\}$ and adds both objects to the intermediate result. It continues to the next sibling subspace $\{mileage, age\}$ which contains four objects (and thus too many to fit the result of required size k). Without selecting any of those objects, the algorithm continues to the next lower subspace $\{mileage\}$ which contains only the object a which is already part of the intermediate result. The same is the case for the next sibling subspace $\{price\}$ and the object $\{b\}$. Finally, object c is added to the result from the next sibling subspace $\{age\}$ and the computation ends.

Fig. 2.9 Telescope on a car database with attributes *mileage*, *price*, *age* and skyline objects $\{a, b, c, d, e\}$

$k:=3$ with precedence
 $\text{mileage} \Rightarrow \text{price} \Rightarrow \text{age}$

Intermediate Results

Step	Result
1	$\{\}$
2	$\{a, b\}$
3	$\{a, b\}$
4	$\{a, b, c\}$



2.6 Cooperative Approaches

The previously presented approaches strictly rely on structural or statistical properties in order to reduce the size of the skyline set. However, all these approaches need to introduce quite momentous assumptions with respect of the implicated semantics (e.g., a skyline object dominating more database objects than another one is better, Pareto-incomparable objects should be treated as dominated under certain conditions, objects occurring in more subspaces are more important, etc.). Thus, although these approaches successfully decrease the size and, in some cases, also improve the computation speed of the reduced skyline set, it remains unclear if the resulting sets are more helpful to the user.

The approaches presented in this section choose a different strategy for choosing objects from an unmanageable large skyline. Instead of relying on assumed structural semantics capturing interestingness, cooperative approaches interactively elicit additional preference information directly from the user in order to steer the selection of Skyline tuples from large result sets.

2.6.1 Interactive Preference Elicitation

Early cooperative approaches like e.g. the online algorithm presented in [3] allowed for a progressive computation of the skyline set, enabling the user to halt the process at any time in order to provide additional or different preferences for further customizing the result set. In contrast, the works in [36] suggest an interactive preference elicitation process suggesting the most informative preference decisions to the user and is geared towards partial ordered preferences on categorical domains. Especially partial order preferences are susceptible to overly large skyline sets, mainly due to incompleteness of the provided user preferences (e.g., isolated values, incomparable values within a single domain, etc.). To address this problem, an iterative elicitation framework is proposed which collects additional

user preferences in each iteration. The framework aims at both minimizing the required amount of user interaction and while also maximizing the resulting skyline reduction. At the same time, it tries to identify a reasonably small and focused skyline set.

This objective is realized by heavily exploiting the knowledge on the cardinality of different domain values with respect to the database instance. During any iteration, the currently available user preferences are examined in context of the current skyline. Based on the distribution of domain values, the algorithm tries to identify those preference statements (i.e., pairs of domain values a and b such that either a preference $a > b$, $a < b$, or $a \sim b$ could be stated by the user) not part of the current user preference which would result in the largest skyline reduction. After identifying the potentially most effective statements in terms of size reduction, they are presented to the user as questions. Now, the user may choose some of the suggested statements which are then added to the user preferences. For example, assume a car database and no previously provided preferences. Assuming that half of the cars are red and the other half is blue, it is sensible to ask the user whether she prefers red or blue cars (or even if she doesn't care about the color). Any decision for either red or blue cars will decrease the skyline significantly. But even if the user is indifferent about the color, the resulting larger skyline set can at least be justified more easily as it can be attributed not to incomplete preference domains but to explicit user feedback.

2.6.2 Trade-Off Skylines

Trade-off skylines [37] approach the problem of overly frequent objects pairs which incomparable by introducing the natural semantics of *trade-offs* or compromises. Trade-offs can be seen as means for compensating between different attribute domains, thus vastly expanding the semantics of simple attribute preferences. However, in contrast to top-k queries [2], the compensation is not of quantitative but of *qualitative* nature.

Consider for example two database objects representing cars: let object A be a 'blue metallic' car for \$18,000 and object B be a 'blue' car for \$17,000, accompanied by a preference favoring cheaper cars and metallic colors. Looking at the ranking on attribute level, both cars are incomparable with respect to the *Pareto order*: one car is cheaper; the other car has the more preferred color. In this scenario, a natural question of a real-life car dealer would be, whether the customer is willing to compromise on those attributes, i.e., if he/she is willing to pay the additional \$1,000 for a metallic paint job for that particular car (such a compromise is called a *trade-offs*). If the answer is yes, then object A is the better choice for the user and should dominate object B with respect to a *trade-off enhanced Pareto order*. However, if some object C like a 'blue' car for \$15,000 exists, A and C would still be incomparable as the premium for the metallic color on that car C is larger than the \$1,000 the user is willing to pay. The basic idea of trade-off skylines is that if users provide several strong trade-offs, many skyline objects can be removed as they are now dominated with respect to this new user feedback. Thus, the skyline is reduced in size and focused consistently with the refined trade-off

enhanced user preferences. Additionally, this kind of user interaction closely models the natural compromises of peoples every day's decision processes. At the same time, the approach abstains from assuming arbitrary user agnostic heuristics for selecting objects.

Trade-offs are elicited interactively, i.e., after computing a preliminary skyline, the user is guided through a trade-off elicitation process which suggests possible effective trade-offs (similar to a car dealer asking his customer additional questions). After the user decides for a trade-off, the trade-off skyline is recomputed and the user interaction continues until the user is satisfied.

However, computing trade-off skylines is quite hard. This is mainly because trade-offs directly modify the product order resulting from attribute preferences which in turn loses its *separability characteristic* [11]. Separability describes the possibility of decomposing the object order losslessly into its respective base preferences (this why most skyline algorithms can avoid operating on the object order at all). In contrast, in [38,39], it was shown that trade-offs will induce additional relationships to the object order, breaking the separability and thus at least materializing some parts of the object order is required. This effect can be explained by the definition of trade-offs: trade-offs can be considered as a user decision between two sample objects focusing on a subspace of the available attributes, while treating all other attributes with *ceteris paribus* semantics. Furthermore, trade-off relationships are transitive and thus may form complex dominance relationships structures spanning several trade-offs and dimensions which are called *trade-off chains*. Especially, the problem of trade-off inconsistencies poses severe challenges as inconsistencies are difficult to detect as they are basically circles in the materialized object order. This problem has been successfully solved in [40].

The algorithms available for computing trade-off skylines have evolved from early algorithms relying on the full materialization of the object order [39], trade-off skylines with severely reduced expressiveness but not requiring the object order at all [41], to computing trade-off skylines allowing the specification of any consistent trade-off without restrictions while still providing acceptable performance by relying on a compressed datastructure minimally representing those parts of the object order which are crucial for computing the trade-off skyline [37].

2.7 Conclusion and Discussion

In this chapter, we have presented different techniques and approaches to address the problem of unmanageable skyline result sets. In general, skyline results have been proven to grow roughly exponential with the number of query dimensions up to result sets containing a significant part of the overall database. Thus, skyline results are usually too large to be useful to users. This effect can be explained by the fairness of the underlying Pareto semantics which cannot establish a sufficient number of dominance relationships between objects in higher dimensions.

One of the central problems for the actual application of skyline queries is thus the question of how to select the most interesting objects from skylines in such a

way that the selection best reflects a user's needs. The resulting selection should be of manageable size, easy to compute, and should contain those objects being most interesting for the user.

The techniques for selecting from skyline sets can be classified in four groups: a) strategies using less strict variations of the Pareto dominance criterion b) approaches aiming at a diverse summarization of the skyline set c) approaches which use additional characteristics of skyline objects or subspace skylines to derive a ranking and selection for the original skyline d) cooperative approaches which elicit additional preference information interactively from the user.

However, please note that each of these presented approaches in some way break the absolute fairness of Pareto semantics and replace them by different heuristics for capturing the notion of a skyline object being "more interesting" than others.

While every presented approach has benefits and advantages on their own right, the imposed heuristics all rely on some "ad-hoc" assumptions on what makes a skyline point more interesting than others. However, the "correctness" and usefulness of these assumptions with respect to the real information needs of a given, individual user is very subjective and thus hard to determine. This issue is especially aggravated by the fact that no approach thoroughly deals with the psychological and perceptive implications of the chosen heuristics. Thus, given a particular application scenario and data source with its user base, it is very hard to decide which approach best tailors to the users' individual personalization requirements, and the decision has to be carefully evaluated by the application designers. Therefore, this chapter is intended to help in choosing a suitable technique for focusing skyline results by summarizing the most prominent techniques and their underlying assumptions.

References

- [1] Börzsönyi, S., Kossmann, D., Stocker, K.: The Skyline Operator. In: *Int. Conf. on Data Engineering (ICDE)*, Heidelberg, Germany (2001)
- [2] Fagin, R., Lotem, A., Naor, M.: Optimal Aggregation Algorithms for Middleware. In: *Symposium on Principles of Database Systems (PODS)*, Santa-Barbara, California, USA (2001)
- [3] Kossmann, D., Ramsak, F., Rost, S.: Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In: *Int. Conf. on Very Large Data Bases, VLDB*, Hongkong, China (2002)
- [4] Papadias, D., Tao, Y., Fu, G., Seeger, B.: An Optimal and Progressive Algorithm for Skyline Queries. In: *International Conference on Management of Data (SIGMOD)*, San Diego, USA (2003)
- [5] Lacroix, M., Lavency, P.: Preferences: Putting More Knowledge into Queries. In: *Int. Conf. on Very Large Data Bases (VLDB)*, Brighton, UK (1987)
- [6] Chan, C.-Y., Eng, P.-K., Tan, K.-L.: Stratified Computation of Skylines with Partially-Ordered Domains. In: *International Conference on Management of Data (SIGMOD)*, Baltimore, USA (2005)

- [7] Bentley, J.L., Kung, H.T., Schkolnick, M., Thompson, C.D.: On the Average Number of Maxima in a Set of Vectors and Applications. *Journal of the ACM (JACM)* 25 (1978)
- [8] Chaudhuri, S., Dalvi, N., Kaushik, R.: Robust Cardinality and Cost Estimation for Skyline Operator. In: 22nd Int. Conf. on Data Engineering (ICDE), Atlanta, Georgia, USA (2006)
- [9] Godfrey, P.: Skyline Cardinality for Relational Processing. In: Seipel, D., Turull-Torres, J.M. (eds.) FoIKS 2004. LNCS, vol. 2942, pp. 78–97. Springer, Heidelberg (2004)
- [10] Balke, W.-T., Zheng, J.X., Güntzer, U.: Approaching the Efficient Frontier: Cooperative Database Retrieval Using High-Dimensional Skylines. In: Zhou, L.-z., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, pp. 410–421. Springer, Heidelberg (2005)
- [11] Hansson, S.O.: Preference Logic. In: *Handbook of Philosophical Logic*, vol. 4, pp. 319–393 (2002)
- [12] Godfrey, P., Shipley, R., Gryz, J.: Algorithms and Analyses for Maximal Vector Computation. *The VLDB Journal* 16, 5–28 (2007)
- [13] Eng, P.-K., Ooi, B.C., Tan, K.-L.: Indexing for Progressive Skyline Computation. *Data 4 Knowledge Engineering* 46, 169–201 (2003)
- [14] Godfrey, P., Gryz, J., Liang, D., Chomicki, J.: Skyline with Presorting. In: 19th International Conference on Data Engineering (ICDE), Bangalore, India (2003)
- [15] Papadias, D., Tao, G.F.Y., Seeger, B.: Progressive Skyline Computation in Database Systems. *ACM Transactions on Database Systems* 30, 41–82 (2005)
- [16] Ciaccia, P., Patella, M., Bartolini, I.: Efficient Sort-Based Skyline Evaluation. *ACM Transactions on Database Systems* 33 (2008)
- [17] Torlone, R., Ciaccia, P.: Finding the Best When It’s a Matter of Preference. In: 10th Italian Symposium on Advanced Database Systems (SEBD), Portoferraio, Italy (2002)
- [18] Boldi, P., Chierichetti, F., Vigna, S.: Pictures from Mongolia: Extracting the Top Elements from a Partially Ordered Set. *Theory of Computing Systems* 44, 269–288 (2009)
- [19] Kim, T., Park, J., Kim, J., Im, H., Park, S.: Parallel Skyline Computation on Multi-core Architectures. In: 25th International Conference on Data Engineering (ICDE), Shanghai, China (2009)
- [20] Fishburn, P.: Preference Structures and Their Numerical Representations. *Theoretical Computer Science* 217, 359–383 (1999)
- [21] Kießling, W.: Foundations of Preferences in Database Systems. In: 28th Int. Conf. on Very Large Data Bases (VLDB), Hong Kong, China (2002)
- [22] Chomicki, J.: Querying with Intrinsic Preferences. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 34–51. Springer, Heidelberg (2002)
- [23] Balke, W.T., Güntzer, U., Siberski, W.: Restricting Skyline Sizes Using Weak PARETO Dominance. *Informatik - Forschung und Entwicklung* 21, 165–178 (2007)
- [24] Kießling, W.: Preference Queries with SV-Semantics. In: 11th Int. Conf. On Management of Data (COMAD 2005), Goa, India (2005)
- [25] Chan, C.-Y.: Finding k-Dominant Skylines in High Dimensional Space. In: ACM SIGMOD Int. Conf. on Management of Data (SIGMOD 2006), Chicago, Illinois, USA (2006)

- [26] Koltun, V., Papadimitriou, C.: Approximately Dominating Representatives. In: Eiter, T., Libkin, L. (eds.) *ICDT 2005*. LNCS, vol. 3363, pp. 204–214. Springer, Heidelberg (2005)
- [27] Yuan, Y.: Efficient Computation of the Skyline Cube. In: *31st Int. Conf. on Very Large Databases (VLDB 2005)*, Trondheim, Norway (2005)
- [28] Pei, J.: Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In: *31st Int. Conf. on Very Large Databases (VLDB 2005)*, Trondheim, Norway (2005)
- [29] Pei, J.: Towards Multidimensional Subspace Skyline Analysis. *ACM Transactions on Database Systems (TODS)* 31, 1335–1381 (2006)
- [30] Chan, C.-Y., Jagadish, H.V., Tan, K.-L., Tung, A.K.H., Zhang, Z.: On High Dimensional Skylines. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) *EDBT 2006*. LNCS, vol. 3896, pp. 478–495. Springer, Heidelberg (2006)
- [31] Vlachou, A., Vazirgiannis, M.: Ranking the Sky: Discovering the Importance of Skyline Points through Subspace Dominance Relationships. *Data & Knowledge Engineering* 69, 943–964 (2010)
- [32] Brin, S., Page, L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems* 30, 107–117 (1998)
- [33] Lin, X., Yuan, Y., Zhang, Q., Zhang, Y.: Selecting Stars: The k Most Representative Skyline Operator. In: *23rd IEEE International Conference on Data Engineering, Istanbul, Turkey (2007)*
- [34] Tao, Y., Ding, L., Lin, X., Pei, J.: Distance-Based Representative Skyline. In: *25th Int. Conf. on Data Engineering (ICDE)*, Shanghai, China (2009)
- [35] Lee, J., You, G.-W., Hwang, S.-W.: Personalized Top-k Skyline Queries in High-Dimensional Space. *Information Systems* 34, 45–61 (2009)
- [36] Lee, J., You, G.-W., Hwang, S.-W., Selke, J., Balke, W.-T.: Optimal Preference Elicitation for Skyline Queries over Categorical Domains. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) *DEXA 2008*. LNCS, vol. 5181, pp. 610–624. Springer, Heidelberg (2008)
- [37] Lofi, C., Güntzer, U., Balke, W.-T.: Efficient Computation of Trade-Off Skylines. In: *13th International Conference on Extending Database Technology (EDBT)*, Lausanne, Switzerland (2010)
- [38] Balke, W.-T., Lofi, C., Güntzer, U.: Incremental Trade-Off Management for Preference Based Queries. *International Journal of Computer Science & Applications (IJCSA)* 4, 75–91 (2007)
- [39] Balke, W.-T., Güntzer, U., Lofi, C.: Eliciting Matters – Controlling Skyline Sizes by Incremental Integration of User Preferences. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) *DASFAA 2007*. LNCS, vol. 4443, pp. 551–562. Springer, Heidelberg (2007)
- [40] Lofi, C., Balke, W.-T., Güntzer, U.: Consistency Check Algorithms for Multi-Dimensional Preference Trade-Offs. *International Journal of Computer Science & Applications (IJCSA)* 5, 165–185 (2008)
- [41] Lofi, C., Balke, W.-T., Güntzer, U.: Efficient Skyline Refinement Using Trade-Offs Respecting Don't-Care Attributes. *International Journal of Computer Science and Applications (IJCSA)* 6, 1–29 (2009)

Chapter 3

Processing Framework for Ranking and Skyline Queries

Seung-won Hwang

Abstract. In the previous chapter, the need to support ranking and skyline queries for multi-criteria decision-making for given user preferences was motivated. We now survey existing algorithms for each query and show a ‘meta-algorithm’ framework for each query. The goal of this chapter is to show that how this framework and cost model enable us to (a) generalize existing algorithms and (b) observe important principles not observed from individual algorithms.

3.1 Introduction

Ranking and skyline queries have gained a lot of attention for multi-criteria decision making in large-scale datasets as Chapter 2 discussed. Ranking enables us to order results based on user preference expressed in numerical *ranking functions*. Skyline query relaxes the requirement by replacing the need for a ranking function with the notion of *dominance*—if a point p is better than another point q in at least one dimension and not worse than q in all other dimensions, it is said that p *dominates* q . Given a multi-dimensional dataset, the skyline query thus returns a subset of “interesting” points that are not dominated by any other points.

To illustrate, we describe an example using an SQL-style expression, as also used in [4, 7].

Example 1 (Ranking and skyline queries). Consider a hotel retrieval system using a database called *Hotel(hno, name, price, distance)*, where *distance* represents the distance from the city center.

Seung-won Hwang
POSTECH, Korea
e-mail: swhwang@postech.edu

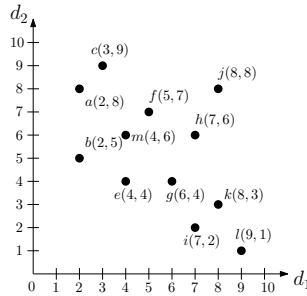


Fig. 3.1 A toy dataset in two dimensional space.

A user preferring a cheap hotel close to the city center may formulate the following ranking query, to find the best hotels.

```
SELECT * FROM Hotel
ORDER BY avg(price, distance)
STOP AFTER 1;           (Q1)
```

Though the above query uses the “average” of *price* and *distance* as a ranking function, the function can be changed to arbitrary monotonic functions to represent different user preferences. For instance, another user may weigh *price* significantly higher, while another may weigh *distance* more. In the toy dataset shown in Figure 3.1, when the average is used as ranking function, hotel like *b*, with equal strength in both *price* (d_1) and *distance* (d_2) will be retrieved. Meanwhile, another user weighing *distance* significantly higher may find hotel *l* more attractive.

However, finding the adequate ranking function itself is a non-trivial problem, which has motivated skyline queries. Skyline queries do not require users to specify ranking functions. Instead, users are only required to specify the attributes they want to minimize (or maximize). The following query illustrates the skyline query counterpart for *Q1*.

```
SELECT * FROM Hotel
SKYLINE OF price MIN, distance MIN;   (Q2)
```

Given these attributes, skyline query algorithms retrieve a set of “desirable” objects, for any monotonic function in general. For instance, in Figure 3.1, hotel *m* is worse than hotel *b* in both aspects, *i.e.*, *m* falls into the shade of *b*’s “dominating region”, *i.e.*, *m* is dominated by *b*. In other words, in any monotonic ranking function, *m* is less preferred than *b*. Skyline results correspond to the objects that are not dominated by any other object, *i.e.*, $\{b, e, i, l\}$ in the figure. These hotels are guaranteed to contain the top hotel for any monotonic ranking function, without requiring users to specify the exact ranking function.

This chapter extends the detailed discussions on existing algorithms for ranking and skyline queries, which can be categorized into *sorted-order* algorithms

that leverage the sorted ordering by attributes (or some aggregate function of attributes) and (2) *partitioning-based* algorithms that focus on dividing the problem space into smaller subspaces for divide-and-conquer optimization.

Given existing algorithms, summarized in Section 3.2, the goal of this chapter is to extend the wisdom of Boolean query optimization [24] for this space. Specifically, to determine the most efficient way to execute a Boolean query, the query optimizer enumerates many possible ways, *e.g.*, query plans, and searches the most “cost-effective” one for the given access scenario. If indices on a query attribute exist in the given scenario, query plans with index will have a lower estimated cost and likely be selected, while the same plan may be disregarded in another scenario. This abstraction, of seeing query optimization as a “cost-guided” search over possible algorithms, has been discussed for both ranking [15] and skyline queries [19].

In this article, we will discuss how such perspectives can complement current solution “spaces” of many hand-crafted algorithms for various access scenarios. Such an approach, if successful, can have the following two advantages, as we witnessed from the same success in Boolean query optimization:

- Meta-algorithm framework: a single implementation can be used for a wide space of access scenarios.
- Systematic optimization: optimization, being systematically guided by the underlying cost model, can often achieve speedups over hand-crafted algorithms.

Toward achieving this goal, we enumerate the following steps:

- Extracting a common skeleton: From a “space” of possible algorithms, a common skeleton can be extracted. For instance, Boolean query processing algorithms are essentially “plan trees” where each node corresponds to different algorithms (*e.g.*, sort-merge join, hash join, nested loop) leveraging the underlying access structure (*e.g.*, index). Similarly, a skeleton can be observed from existing ranking and skyline algorithms. Then any algorithm generated from this skeleton is a potential candidate algorithm, which essentially enumerates a “space” of possible solutions.
- Implementing an efficient search: Once the space is defined, query optimization is essentially a search problem of finding the cost-optimal algorithm for the given scenario. However, typically, the search space is too large and a naive search for an efficient algorithm can easily outweigh the cost of executing algorithms, which defeats the whole purpose. To keep optimization overhead reasonable, techniques, such as reducing the search space without eliminating good candidate algorithms (*e.g.*, considering only left-deep joins in Boolean queries), should be developed.

This article will discuss how the above steps have been taken for ranking and skyline queries respectively.

The rest of this article is organized as follows. Section 3.2 reviews existing algorithms. Section 3.3 discusses the unified ranking framework and how it

benefits existing ranking algorithms, and Section 3.4 discusses the same for skyline queries. Finally, Section 3.5 concludes this article.

3.2 Related Work

Ranking and skyline algorithms can be categorized by many criteria, but in this chapter, we focus on the scope of (a) sequential algorithms (b) categorized by sorted-order and partitioning-based algorithms. For a taxonomy and survey with larger coverage and diverse criteria, refer to [16] (ranking queries) and [12] (skyline queries).

3.2.1 Sorted-Order Algorithms

Ranking algorithms depend on pre-sorted ordering, known as “sorted access” of objects by some attribute. Pioneering ranking algorithms TA [10] and FA [11] use sorted accesses. Each of these algorithms materializes the sorted ordering on each query attribute, and aggregate them into the overall ranking order. These algorithms require pre-sorted ordering materialized for all attributes, *e.g.*, using indices on local databases.

However, in some scenarios, such sorted access may not be available or may be very expensive to access. Algorithm MPro [6] is specifically designed for such scenarios where sorted access is unavailable, such as when attribute value is determined by querying external sources (or determined dynamically by some user-defined function). In this case, algorithms should rely on alternative access mode, *i.e.*, “random access”. Meanwhile, in other scenarios, some attribute may support both sorted and random access, but random access is significantly more expensive, for which a batch of algorithms have been hand-crafted, *e.g.*, CA [11] and Stream-Combine [14].

For skyline query processing, Balke et al. [2] proposed an algorithm using all attribute orderings for distributed skyline computation. Further, Lo et al. [21] proposed a progressive skyline computation under the same model and studied how to find a good “terminating object” using a linear regression technique for early termination. Meanwhile, Chomicki et al. [8, 9] proposed SFS, pre-materializing a single aggregated ordering of data, using some monotone ranking function. Later, Godfrey et al. [12] developed LESS as a combination of both BNL and SFS, which accesses points in stored order (as in BNL) but keeping skyline candidates in the order of dominance regions (as in SFS) to reduce the dominance tests for non-skyline points. Recently, Bartolini et al. [3] enhanced SFS by using *minC* function to achieve early termination in distributed data scenarios.

Table 3.1 Taxonomy of existing ranking algorithms.

All sorted accesses	Part	None
FA [10], TA [11], Quick-Combine [13] (cheap random)	MPro [6]	NRA [11]
CA [11] SR-Combine [1] (expensive random)	Upper [5]	Stream-Combine [14]

Table 3.2 Taxonomy of existing skyline algorithms.

Sorted-order	Partitioning-based
BNL [4], SFS [8, 9], LESS [12], SaLSa [3], SSkyline [?]	D&C [4], NN [17], BBS [23], LS [22], ZSearch [20] OSPS [25], SkyTree [18]

3.2.2 Partitioning-Based Algorithms

Alternatively, ranking algorithms may use multi-dimensional indices partitioning data space for efficient rank query processing. Similarly, partitioning-based algorithms can be devised for skyline queries, grouping points into subregions that share a commonality to carry out region-based dominance tests. An early algorithm, D&C [4] simply divided the problem into multiple sub-problems and merged the local skyline points into a global skyline. To use more efficient region-level access, NN [17] and BBS [23] built upon pre-constructed spatial indices like the R-tree. Recently, Lee et al. [20] proposed ZSearch using *ZB-tree* as a new variant of *B-tree*, and Morse et al. [22] proposed LS using a static lattice structure for the special case of low-cardinality datasets.

Partitioning algorithms do not necessarily build upon indices. There are existing algorithms that partition data at the run time, *e.g.*, Zhang et al. [25] and Lee et al. [18] proposed partitioning-based algorithms without pre-computed indices. These algorithms outperformed sorting-based algorithms, by considering both dominance and incomparability. However, no partitioning-based ranking algorithm is a “non-index” algorithm, as ranking algorithms without any prematerialized access structure are at best sequential scans.

3.2.3 Taxonomy and Generalization

We now put the algorithms mentioned above in the taxonomy of ranking and skyline algorithms. We then discuss a unified framework for each taxonomy.

Ranking algorithms: Table 3.1 summarizes a “matrix” of all possible scenarios and shows existing “sorted-order” ranking algorithms (as not many partitioning-based ranking algorithms exist.) Depending on the scenario, the given algorithm leverages all, part, or no sorted-ordering on attributes.

Skyline algorithms: Table 3.2 summarizes existing skyline algorithms in the taxonomy of sorted-order and partitioning-based algorithms.

Meta-algorithm framework: For SQL queries, a standard optimization framework [24] abstracts possible algorithms as multiple query plans and performs a cost-driven search for a good query plan. As the space of possible plans is inherently large, optimizer does not aim at enumerating all plans and selecting the absolute best. Instead, the optimizer aims at reducing the space (without eliminating good algorithms much) to select one among low cost candidates with good trade-off between the time spent on finding a good plan and time running such a plan.

Similar approach has been taken for ranking [15] and skyline [19] queries. Similarly, such optimizers first define a space of algorithms and develop optimization techniques to guide the search to low-cost algorithms. In this chapter, we will discuss these two works in detail, in Section 3.3 and 3.4 respectively, to show how extending the wisdom of SQL query optimization complements existing state-of-the-arts in both problems and opens up new avenues for further optimization.

3.3 Framework for Ranking Queries

This section first discusses preliminaries, then extracts a common discusses cost-driven optimization strategies for ranking queries.

3.3.1 Preliminaries

We first introduce some basic notations to formally present ranking and skyline queries. Let \mathcal{D} be a finite d -dimensional space, *i.e.*, $\{d_1, \dots, d_d\}$, where each dimension has a domain of nonnegative real number \mathbb{R}^+ , denoted as $dom(d_i) \rightarrow \mathbb{R}^+$. Let \mathcal{S} be a set of finite n points that is a subset of $dom(\mathcal{D})$, *i.e.*, $\mathcal{S} \subseteq dom(\mathcal{D})$. A point p in \mathcal{S} is represented as (p_1, \dots, p_d) in which $\forall i \in [1, d] : p_i \in dom(d_i)$. For simplicity, $dom(d_i)$ has normalized values of $[0, 1]$.

Based on these notations, we formally define *monotonicity* and *ranking queries*.

Definition 1 (Monotonicity). Given $p, q \in \mathcal{S}$, ranking function \mathcal{F} is monotonic, $\mathcal{F}(p) \leq \mathcal{F}(q)$ if $\forall i \in [1, d] : p_i \leq q_i$.

	d_1	d_2	\mathcal{F}
a	2		
b	2		
c	3		
m	4		
e	4	4	4
g		4	
i		2	
k		3	
l		1	

Fig. 3.2 Intermediate table for FA.

	d_1	d_2	\mathcal{F}
a	2	8	5
b	2	5	3.5
c	3		
e		4	
g		4	
i	7	2	
k	8	3	
l	9	1	

Fig. 3.3 Intermediate table for A.

Definition 2 (Ranking). Given user-specified retrieval size k and ranking function \mathcal{F} , correct top- k results is a set of points such that $\mathcal{K}_{\mathcal{D}}(\mathcal{S}) = \{p \in \mathcal{S} \mid \nexists q \in \mathcal{S} - \mathcal{K} : \mathcal{F}(q) \geq \mathcal{F}(p)\}$ and $|\mathcal{K}_{\mathcal{D}}(\mathcal{S})| = k$.

We now describe how existing ranking algorithms are designed, using Q_1 in Section 3.1 as a running query, on the hotel data set described in Figure 3.1. This query asks for the top $k = 1$ hotel with the smallest $avg(d_1, d_2)$. Recall that, existing algorithms build upon the following two access modes of *sorted* access and *random* access.

Example 2 (Example Algorithm). In this example, we describe FA, known as a pioneering algorithm, for finding the top-1 hotel. This algorithm uses sorted access on d_1 and d_2 , which materialize the hotels in ascending order of each attribute values, *i.e.*, $a, b, c, m, e, f, g, h, i, j, k, l$ (ties broken in alphabetical order) for d_1 and $l, i, k, e, g, b, m, h, f, a, j, c$ for d_2 . At each iteration, FA retrieves one object from each list using *sorted access*, *e.g.*, a and l in the first iteration, which continues until the same object is found in both lists. Specifically, after five iterations, e is retrieved from both lists and the attribute values accessed from the preceding five iterations are shown in Table 3.3. This is the point where FA safely concludes that the top-1 hotel is among 9 hotels in Table 3.3, even though the full \mathcal{F} score is known only for e . That

is, any hotel that does not appear in this table, *e.g.*, hotel f , cannot score higher than e , as both f_1 and f_2 values are higher than e_1 and e_2 (as the ranking function is monotonic). However, this does not mean we can conclude e as the top hotel, as other 8 hotels in the table still have the chance to outscore e . To eliminate such possibilities, we use random access to fill in the unknown scores, *e.g.*, random access on a_2 . Once the Table 3.3 is completely filled, we can return the result with the smallest average, which is hotel b with average 3.5.

The above described algorithm, known as FA, can be represented by the sequence of accesses it performs. When denoting sorted access on d_i as sa_i and random access on d_i value of hotel a as ra_i^a , the above algorithm executes the following sequence:

$(sa_1, sa_2, sa_1, sa_2, sa_1, sa_2, sa_1, sa_2, sa_1, sa_2, ra_2^a, ra_2^a, ra_2^b, ra_2^c, ra_2^m, ra_1^g, ra_1^i, ra_1^k, ra_1^l)$. However, this is just one algorithm and there can be many more algorithms, differing in the access sequences they perform. To illustrate, consider an alternative algorithm A , performing deeper sorted access on d_2 :

$(sa_1, sa_1, sa_1, sa_2, sa_2, sa_2, sa_2, sa_2, ra_2^a, ra_2^b, ra_1^i, ra_1^k, ra_1^l)$.

This algorithm can also return b as the result, as unknown d_2 values are no less than 4 (if less, those values should have been accessed from preceding sorted access) and unknown d_1 values no less than 3. This suggests that the average of all the hotels in the table (and those now shown in the table) will be no less than b .

Different algorithms, by performing different accesses in different orders, incur different costs. For instance, A can answer the same query, performing only a part of accesses that Algorithm FA performs, *i.e.*, A is better than FA in all scenarios. In some cases, two algorithms A_1 and A_2 may have different strengths. A_1 performing more sorted access but less random access than A_2 . In that case, A_1 is superior in some scenarios while inferior in others.

In most ranking algorithms, cost function \mathcal{C} sums up the unit access cost of sorted and random access performed, which we denote as cs_i and cr_i . That is:

$$\mathcal{C} = \sum_i cs_i \times ns_i + cr_i \times nr_i \quad (3.1)$$

where ns_i and nr_i denote the number of random and sorted access the given algorithm performs on d_i .

3.3.2 Extracting Skeleton

Toward the goal of finding the optimal algorithm in the given scenario, [15] observed a common skeleton from ranking algorithms. As different algorithms essentially differ in terms of the access sequence, any such algorithm can be generated from the following skeleton:

```

while (unsafe to terminate)
    select  $a$  from any possible accesses  $sa_i$  and  $ra_i$ ;
    perform  $a$ ; update score table;

```

Fig. 3.4 Algorithm “skeleton”.

With this skeleton, reference [15] abstracts query optimization as searching with the minimal cost \mathcal{C} , from a “space” Ω of possible algorithms, that can be generated from the above skeleton. Formally,

$$A_{opt} = \operatorname{argmin}_{A \in \Omega} \mathcal{C}(A)$$

However, the framework renders too many algorithms, *i.e.*, all possible sequences of sorted/random access, which discourages runtime query optimization. We thus follow the wisdom of SQL optimization, balancing the trade-off between time spent on searching for a plan and time spent on running a plan. Specifically, the SQL optimizer does the following:

- **Space reduction:** Instead of considering the entire solution space, the optimizer consider a smaller subset. However, to avoid throwing out low-cost plans, ideally, for every algorithm *out* of the chosen subset, there should be a counterpart algorithm *in* the space doing the same job with no higher cost. An example of space reduction for SQL query processing is considering left-deep joins (and throwing out right-deep or bushy joins), as a good right-deep or bushy plan has its left-deep equivalent in the subset, which will be considered in optimization.
- **Efficient search:** Even with this reduced space, performing an exhaustive search incurs prohibitive cost. To expedite a search, SQL query optimizer uses dynamic programming or simulated annealing to narrow down to a low-cost plan fast.

```

while (unsafe to terminate)
    perform  $sa_i$  until the value retrieved gets higher than  $\delta_i$ ;
    perform  $ra_i^o$  on object  $o$  with highest upper bound score

```

Fig. 3.5 Skeleton generating a reduced space.

3.3.3 Cost-Based Optimization Strategies

Strategies to reduce the search space, driven by the cost function \mathcal{C} have been extended to ranking queries in [15]. Specifically, the paper formally discusses how the space can be significantly reduced without compromising the quality of the algorithm much. In particular, Figure 3.5 illustrates a new skeleton generating a much reduced space, which performs all the necessary sorted accesses (up to certain depth δ_i), before any random access. With this skeleton, different algorithms with varying δ are generated with different costs.

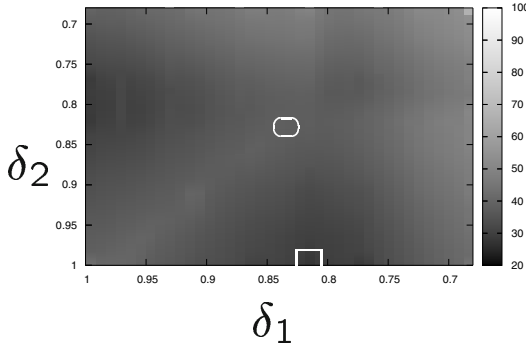


Fig. 3.6 Illustration of solution space [15].

To illustrate this problem space, Figure 3.6 shows the costs of algorithms generated by different (δ_1, δ_2) value pairs (darker cells indicate low cost). In this figure, the goal of optimization is finding the darkest cell, as shown in the rectangle (a cell in circle corresponds to TA).

For an efficient search for the lightest cell, a hill climbing approach was used in [15]. That is, in Figure 3.6, once this reduced space was finalized, the paper develops a hill-climbing search, which starts with $\delta_1 = \delta_2 = 1$ and tries neighboring algorithms with lower estimated cost next, which terminates when no neighboring algorithm incurs lower cost. Comparing the algorithm found from cost-based framework (in rectangle) with TA (in circle) provides an interesting insight: TA, by being hard-wired to perform round-robin sorted access, focuses on looking for algorithms in the diagonal, which is not always effective. Instead, cost-driven hill-climbing allows us to generate an algorithm like the one in rectangle, performing sorted access mostly on one attribute,

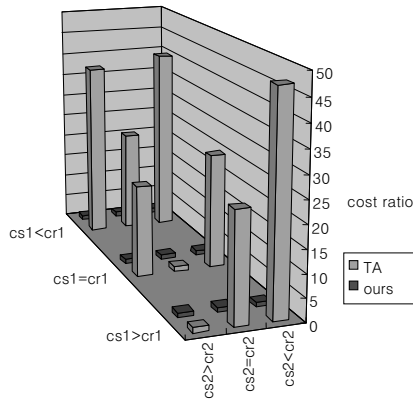


Fig. 3.7 TA vs cost-based reported in [15].

which would be effective if the sorted access cost of that attribute is much cheaper.

Due to this “generality”, according to empirical results reported in [15], which we show here in Figure 3.7, unlike TA showing strength in some scenarios (where round-robin sorted access is effective), cost-based optimization performs well steadily over all the scenarios. However, in the scenarios where TA performs well, cost-based approach performs comparably well (though it incurs additional overhead for the solution search). For the details on experimental setting, refer to [15].

3.4 Framework for Skyline Queries

This section first discusses preliminaries, then extracts a common discusses cost-driven optimization strategies for skyline queries.

3.4.1 Preliminaries

We first formally define *dominance*, *incomparability*, and *skyline queries* respectively. These definitions are consistent with existing skyline work. Throughout this paper, we consistently use *min* operator for skyline computation.

Definition 3 (Dominance). Given $p, q \in \mathcal{S}$, p dominates q on \mathcal{D} , denoted as $p \prec_{\mathcal{D}} q$, if and only if $\forall i \in [1, d] : p_i \leq q_i$ and $\exists j \in [1, d] : p_j < q_j$.

Definition 4 (Incomparability). Given $p, q \in \mathcal{S}$, p and q are *incomparable* on \mathcal{D} , denoted as $p \sim_{\mathcal{D}} q$ if and only if $p \not\prec_{\mathcal{D}} q$ and $q \not\prec_{\mathcal{D}} p$.

Definition 5 (Skyline). A point p is a *skyline point* on \mathcal{D} if and only if any other point $q (\neq p)$ does not dominate p on \mathcal{D} . Given dataset \mathcal{S} on \mathcal{D} , *skyline* is a set of skyline points such that $\text{SKY}_{\mathcal{D}}(\mathcal{S}) = \{p \in \mathcal{S} \mid \nexists q \in \mathcal{S} : q \prec_{\mathcal{D}} p\}$.

For later uses, we straightforwardly extend the notion of *dominance* and *incomparability* to “region-level” notions. Suppose that a hyper-rectangle region R on \mathcal{D} is represented as $[u_1, v_1] \times \dots \times [u_d, v_d]$ in which a virtual best point and a virtual worst point are denoted as $R_b = (u_1, \dots, u_d)$ and $R_w = (v_1, \dots, v_d)$ respectively. For this regional unit, we formally present the notions of dominance and incomparability, as done in [20].

Definition 6 (Region-level dominance). Given two regions R and R' on \mathcal{D} , R dominates R' if $R_w \prec_{\mathcal{D}} R'_b$.

Definition 7 (Region-level incomparability). Given two regions R and R' on \mathcal{D} , they are *incomparable* if $R_b \not\prec_{\mathcal{D}} R'_w$ and $R'_b \not\prec_{\mathcal{D}} R_w$.

For brevity of representation, we replace notations $\prec_{\mathcal{D}}$, $\not\prec_{\mathcal{D}}$, $\sim_{\mathcal{D}}$, and $\text{SKY}_{\mathcal{D}}(\mathcal{S})$ with \prec , $\not\prec$, \sim , and $\text{SKY}(\mathcal{S})$ if it is clear from the context.

As for ranking queries, we aim at finding the optimal algorithm with minimal cost \mathcal{C} . However, the cost function for skyline algorithm is different from that of ranking algorithms:

$$A_{opt} = \operatorname{argmin}_{A \in \Omega} \mathcal{C}(A)$$

Unlike ranking algorithms incurring a sub-linear scan and low computation cost, skyline algorithms require many pairwise dominance tests. Consequently, **cost function \mathcal{C} for skyline algorithms typically counts the number of dominance tests performed.**

Toward this goal, existing algorithms iteratively select a point that dominates a large number of points, *e.g.*, b dominating all the points in the shaded region in Figure 3.8(b). However, we observe these “dominance-based” algorithms suffer the following limitations:

Incomparability is critical to achieve scalability. While existing skyline algorithms have focused mostly on reducing point-wise dominance tests, *incomparability*, *i.e.*, two points p and q not dominating each other, is another key factor in optimizing skyline computation. In particular, in high-dimensional space, most point pairs become incomparable. To illustrate this, Figure 3.11 depicts the number of point pairs with dominance and incomparability. The dataset was synthetically generated with 200,000 uniformly distributed points with independent attributes, and was used as a synthetic dataset. The numbers in parentheses indicate the average number of skyline points. It is clear that, in low dimensionality, almost all of the point pairs have dominance relations. In contrast, for a dimensionality higher than 13, incomparable pairs start to dominate. The graph empirically demonstrates that, to enable skyline query processing scalable over dimensionality, considering both the dominance and incomparability is crucial.

Balancing dominance and incomparability is non-trivial. To optimize both dominance and incomparability, the pivot point selection should be carefully

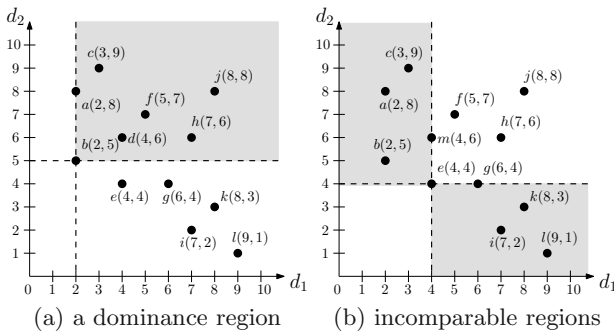


Fig. 3.8 Dominance vs. incomparability.

designed for a balanced optimization of these two factors. Existing work mostly optimized for dominance by choosing a pivot point that maximized the dominance region as depicted in Figure 3.8(a). In clear contrast, an alternative extreme is to optimize solely for incomparability by picking a pivot point e that “evenly” partitions the entire region as shown in Figure 3.8(b) in order to maximize incomparable subregions (marked by shaded rectangles). With this alternative pivot selection, even though the dominance region is reduced, we can still bypass nine dominance tests on point pairs across the two incomparable regions, $\{a, b, c\}$ and $\{i, k, l\}$, as they are guaranteed to be incomparable. However, it is non-trivial to find a *cost-optimal* strategy between these two extremes.

In a clear contrast, cost-based optimization approach enables to balance the two, as we discuss in further details below, guided by the cost function, as we will further elaborate in the next section.

First, region-level dominance is used for reducing dominance tests of non-skyline points. If R dominates R' , a point p in R dominates all points in R' after only one dominance test. This property intuitively inspires the idea of index-based algorithms [17, 23] by pruning a group of points in R' , *i.e.*, an MBR, by a single dominance test between p and R'_b . Similar intuitions are used in non-index algorithms [8, 9, 12, 3] to access pivot point maximizing dominance regions first. These points would be effective in pruning out points in its dominance region early on.

Second, region-level incomparability is used to avoid dominance tests between incomparable points (not necessarily skyline points). Suppose that two regions R and R' are incomparable. In this case, the two points corresponding to each region are also incomparable, *i.e.*, if $R \sim R'$, then $\forall p \in R, \forall q \in R' : p \sim q$. We can thus save computation cost by bypassing dominance tests for point pairs between incomparable regions. This property inspires the idea of partitioning-based algorithms [25, 18], using heuristic pivot selection for incomparability. Recall that, optimizing for incomparability becomes more and more critical in high-dimensional space, where most point pairs become incomparable.

3.4.2 Extracting Skeleton

This section discusses common skeletons of a class of existing algorithms \mathcal{A} from which common modules can be identified as in Figure 3.10. As some algorithms can leverage indices and some cannot, for fair comparison, we assume no pre-constructed index for all algorithms. In these skeletons, two key common modules, **SelectPivot()** and **Prune()**, are identified.

The common optimization goal of all algorithms is to design **SelectPivot()** module, to minimize the cost in **Prune()** module. As dominance tests are the dominating factor in the overall cost (as argued in [20, 25]), we aim at

Algorithm 1. $\text{Opt}(p^V, \mathcal{S})$

```

1:  $\mathcal{S} \leftarrow \text{MapPointToRegion}(p^V, \mathcal{S})$ .
2: // Remove all points in  $S_{2^d-1}$  dominated by  $p^V$ .
3:  $\mathcal{S} \leftarrow \mathcal{S} - \text{Dominance}(p^V, S_{2^d-1})$ .
4:  $\mathcal{B} \leftarrow \{B_0, \dots, B_{2^d-2}\}$ .
5: for  $\forall (B_i, B_j) \in \mathcal{B} \times \mathcal{B}$  do
6:   // Check partial dominance, and remove dominated points.
7:   if  $B_i \prec_{\text{Par}} B_j$  and  $S_i \neq \{\}$  then
8:      $\mathcal{S} \leftarrow \mathcal{S} - \text{Dominance}(S_i, S_j)$ .
9:   else if  $B_i \sim B_j$  then
10:    Continue. // Skip dominance tests between  $S_i$  and  $S_j$ .
11:   end if
12: end for
13: return  $\mathcal{S}$ 

```

developing **SelectPivot()** module that minimizes the number of dominance tests in **Prune()** module.

Toward this goal, an optimal implementation **Opt** of **Prune()** can be defined, as shown in Algorithm 1, requiring minimal dominance tests to find correct skyline results. More formally, given an algorithm A and a dataset \mathcal{S} , let us denote the cost model as $\mathcal{C}(A, \mathcal{S})$. The optimality of **Opt** can be stated as follows:

For the sake of explaining **Opt** and proving its optimality, we define the following “lattice” structure.

Given a pivot point p^V , the entire region can be divided into disjoint 2^d subregions. For instance, Figure 3.8(b) describes four subregions partitioned by a point e . Every point in \mathcal{S} is thus contained in a subregion, e.g., $\{\}$, $\{a, b, c\}$, $\{i, k, l\}$ and $\{m, e, f, g, h, j\}$. Formally, let \mathcal{R} denote a set of subregions on \mathcal{D} , i.e., $\mathcal{R} = \{R_0, \dots, R_{2^d-1}\}$, based on which we represent region-level relations.

More specifically, to simplify region-level relations, we introduce a d -dimensional vector B that corresponds to R . Let \mathcal{B} denote a set of all d -dimensional vectors i.e., $\mathcal{B} = \{B_0, \dots, B_{2^d-1}\}$, where each vector is mapped into a subregion such that $\forall B_i \in \mathcal{B} : B_i \rightarrow R_i$. Let $B.d_i$ denote a value on d_i of vector B , where a value d_i on B corresponds to the i^{th} most significant bit. Formally, given a pivot point p^V and a point q in R_i , $B.d_i$ is represented as a binary value:

$$B.d_i \leftarrow \begin{cases} 0, & \text{if } q_i < p_i^V; \\ 1, & \text{otherwise.} \end{cases}$$

We then explain how to infer region-level relations from binary vectors presenting subregions. For instance, when $d = 3$, p^V divides the entire region into eight subregions, i.e., $\mathcal{B} = \{B_0 = 000, B_1 = 001, \dots, B_7 = 111\}$. If $B.d_i$ is 0, the range of possible point values is $[0, p_i^V)$. Otherwise, the range is $[p_i^V, 1]$. Thus, vectors B_0 and B_1 describe subregions $[0, p_1^V) \times [0, p_2^V) \times [0, p_3^V)$

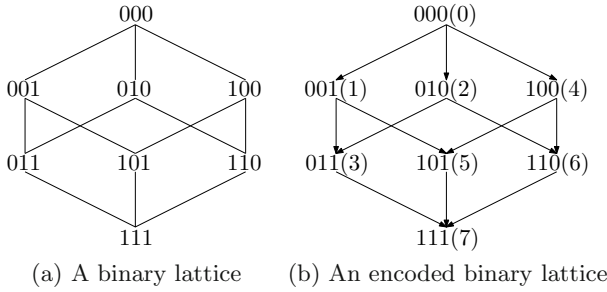


Fig. 3.9 A lattice for mapping points into regions when $d = 3$.

and $[0, p_1^V) \times [0, p_2^V) \times [p_3^V, 1]$ respectively. The binary vectors are thus suited for presenting subregions concisely.

Using binary vectors, region-level relations can be formally represented as the following three cases:

Definition 8 (Dominance in \mathcal{B}). Given two vectors B and B' , B dominates B' on \mathcal{D} , denoted as $B \prec B'$, if and only if $\forall i \in [1, d] : B.d_i < B'.d_i$, i.e., $\forall i \in [1, d] : B.d_i = 0$ and $B'.d_i = 1$.

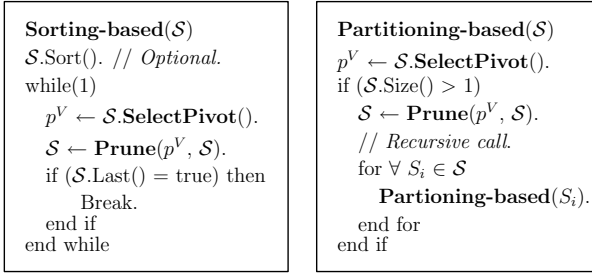
Definition 9 (Partial dominance in \mathcal{B}). Given two vectors B and B' , B partially dominates B' on \mathcal{D} , denoted as $B \prec_{Par} B'$, if and only if $\forall i \in [1, d] : B.d_i \leq B'.d_i$ and $\exists j \in [1, d] : B.d_j = B'.d_j$.

Definition 10 (Incomparability in \mathcal{B}). Given two vectors B and B' , B is incomparable with B' on \mathcal{D} , denoted as $B \sim B'$, if and only if $\exists i \in [1, d] : B.d_i < B'.d_i$ and $\exists j \in [1, d] : B'.d_j < B.d_j$.

The relations between binary vectors can thus be organized as a *partially ordered set*, represented as a *lattice*. To illustrate this, Figure 3.9 describes a binary lattice and its binary encoding when $d = 3$. In this lattice, adjacent node pairs connected by an arrow represent partial dominance relations. By the *transitivity*, node pairs reachable by a path of multiple paths also have partial dominance relations. Among these partially dominated pairs, $(B_0, B_{2^{d-1}})$ shows a dominance relation according to Definition 8. All of the remaining non-reachable pairs have incomparability relations.

More specifically, for all relations such that $\forall i, j \in [0, 2^d - 1] : (B_i, B_j) \in \mathcal{B} \times \mathcal{B}$, we explain how region-level relations are related to point-wise dominance tests. For the sake of representation, let S_i denote a subset of points in R_i mapped into corresponding vector B_i .

- **Dominance:** For dominance pair $(B_0, B_{2^{d-1}})$, if a point p in S_0 exists, any point q in $S_{2^{d-1}}$ can be immediately pruned out after a point-wise dominance test on (p, q) .



(a) Sorting-based scheme (b) Partitioning-based scheme

Fig. 3.10 Skeletons of non-index skyline algorithms.

- **Partial dominance:** The relation can be classified into two subcases: (1) a self-pair (B_i, B_i) and (2) (B_i, B_j) with an arrow from B_i to B_j . Guided by these region-level relations, we then perform point-wise dominance tests to effectively identify point-level relations. First, for self-pairs B_i and B_i , we perform dominance tests in S_i itself. Second, for B_i and B_j with partial dominance relations, since points in S_i are likely to dominate those in S_j , we perform dominance tests between S_i and S_j . Recall that, we take the transitivity into account to find all of the partial dominance relations—If an arrow exists from B_i to B_j and an arrow from B_j to B_k , then B_i also partially dominates B_k . We thus perform dominance tests for partial dominance pairs (B_i, B_k) .
- **Incomparability:** For all the remaining pairs, B_i and B_j are incomparable, which suggests that point sets S_i and S_j in corresponding regions R_i and R_j are also incomparable, *i.e.*, if $B_i \sim B_j$, then $\forall p \in S_i, q \in S_j : p \sim q$. We can thus bypass point-wise dominance tests between S_i and S_j .

We now prove that **Opt** only requires minimal dominance tests to find correct skyline results, by showing that the cost of **Opt** is no higher than that of any arbitrary algorithm $A \in \mathcal{A}$. Suppose that the same pivot is used. Given an algorithm A and a dataset \mathcal{S} , let us denote the cost model as $\mathcal{C}(A, \mathcal{S})$. We formally state the optimality of **Opt** as follows:

Theorem 1 (Optimality of Opt). *Given a set of any non-index algorithms \mathcal{A} , Algorithm **Opt** incurs the minimal cost, *i.e.*, $\forall A \in \mathcal{A} : \mathcal{C}(\mathbf{Opt}, \mathcal{S}) \leq \mathcal{C}(A, \mathcal{S})$.*

Proof. We prove this by contradiction. In other words, if A skips any dominance test performed by **Opt**, it may no longer guarantee that A finds correct skyline results. Specifically, we shows the dominance tests of **Opt** for the following three region-level relations described in the lattice.

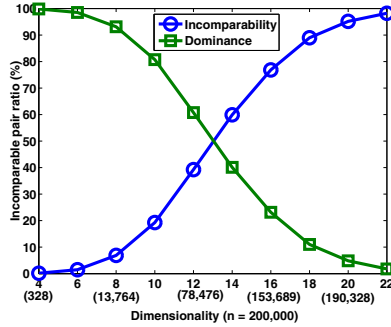


Fig. 3.11 The effect of incomparability.

- **Dominance (lines 2-3):** When performing dominance tests between a point $q \in S_{2^d-1}$ and a pivot point p^V , Opt only requires one dominance test for each point. For S_{2^d-1} , the cost of Opt thus equals $|S_{2^d-1}|$. To contradict, assume that an algorithm A exists with fewer dominance tests. For a skipped dominance test, A can include $q \in S_{2^d-1}$ as final skyline if q is only the point dominated by p^V . Consequently, A causes incorrect results, which incurs a contradiction.
- **Partial dominance (lines 6-8):** Opt needs to check dominance tests between p and q if S_i and S_j have a partial dominance relation. To contradict, assume that A can skip dominance tests between p and q . In this case, A can contain q as final skyline if q is the only point dominated by p . As a result, the result of A is incorrect, which incurs a contradiction.
- **Incomparability (lines 9-10):** Given two regions R_i and R_j , Opt bypasses all the point-wise dominance tests corresponding to S_i and S_j . In this case, A can save as many equal dominance tests as Opt.

To sum up, an algorithm A performing fewer point-wise dominance tests cannot guarantee correct skyline results. In other words, a non-index algorithm has to perform at least as many dominance tests as Opt.

3.4.3 Cost-Based Optimization Strategies

This section designs a systematic pivot point selection. An ideal pivot point maximizes the power of dominance and incomparability. Formally, at each iteration, a pivot point p^V maximizing both $N_D(p^V, \mathcal{S})$ and $N_I(p^V, \mathcal{S})$ should be selected:

$$\begin{aligned}
 p^V &= \operatorname{argmin}_{p \in \mathcal{S}} \mathcal{C}(p, \mathcal{S}). \\
 &= \operatorname{argmax}_{p \in \mathcal{S}} N_D(p, \mathcal{S}) + N_I(p, \mathcal{S}).
 \end{aligned}$$

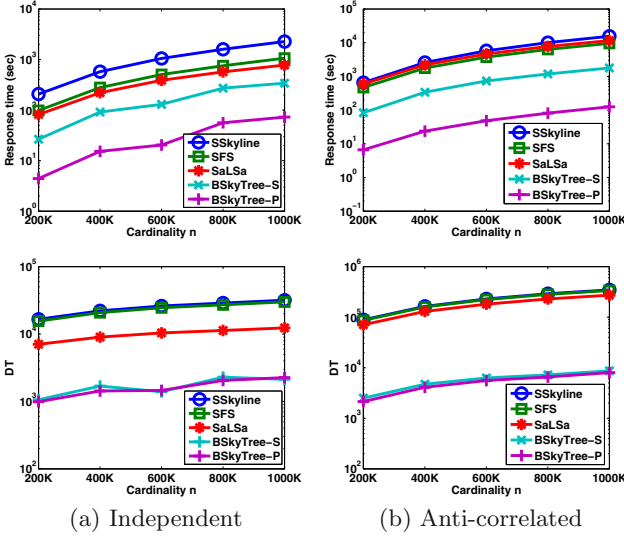


Fig. 3.12 Varying dataset size.

A naive solution to this cost-based optimization would be to compute the function for every point, which would involve performing dominance tests across all pairs of $2^d - 1$ subregions. This naive solution thus incurs a prohibitive cost of $O(4^d)$. As a result, estimating the exact function scores defeats the whole purpose of optimizing the pivot point selection.

As one way to handle this challenge, an effective approximation scheme has been developed in [19], which interleaves the following two phases.

1. **Pruning phase:** To maximize N_D , a dataset is scanned once to consider skyline points with high dominance power as candidates.
2. **Optimization phase:** These candidates are then checked to see if N_I is high as well. Specifically, the number of incomparable pairs is estimated to pick the candidate with the highest N_I .

This “balanced” pivot selection scheme *Balanced*, can then be plugged into the skeleton of sorted-order and partitioning-based algorithms respectively, to generate two new algorithms named *BSkyTree-S* and *BSkyTree-P* respectively. These algorithms, guided by systematic observations and cost function, are empirically reported in [19] to outperform the state-of-the-art non-index algorithms up to two orders of magnitude. Figure 3.12 shows evaluation results from [19] on 12-dimensional data with independent and anti-correlated distribution (for further details on experiment settings, refer to the paper.)

3.5 Conclusion and Open Issues

This chapter briefly surveyed a class of existing algorithms for supporting ranking and skyline queries and discussed how having a mega-algorithm framework can help us to generalize the problem, both ranking and skyline query processing, and observe new principles neglected in the existing efforts. The key contribution of this chapter is to suggest a methodology, which can similarly applied for different classes of algorithms.

As a future direction, one can investigate if the same mega-algorithm framework would benefit related problems as well, *e.g.*, skycube creation. More specifically, existing skycube algorithms identify skyline results for all possible subspaces efficiently, by reusing the “results” from the computation for another subspace. However, considering point-based partitioning results can be shared between two subspaces, one can consider reusing this partitioning “structure” across subspaces, to enable further saving. Similarly, one can investigate, whether this mega-algorithm framework can apply for different cost scenarios, such as parallel processing.

References

1. Balke, W., Guentzer, U., Kiessling, W.: On Real-time Top-k Querying for Mobile Services. In: CoopIS 2002 (2002)
2. Balke, W.-T., Güntzer, U., Zheng, J.X.: Efficient Distributed Skylining for Web Information Systems. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 256–273. Springer, Heidelberg (2004)
3. Bartolini, I., Ciaccia, P., Patella, M.: Efficient Sort-Based Skyline Evaluation. ACM TODS (2008)
4. Börzsönyi, S., Kossmann, D., Stocker, K.: The Skyline Operator. In: ICDE (2001)
5. Bruno, N., Gravano, L., Marian, A.: Evaluating Top-k Queries over Web-Accessible Databases. In: ICDE (2002)
6. Chang, K.C.C., Hwang, S.: Minimal Probing: Supporting Expensive Predicates for Top-k Queries. In: SIGMOD 2002 (2002)
7. Chaudhuri, S., Dalvi, N., Kaushik, R.: Robust Cardinality and Cost Estimation for Skyline Operator. In: ICDE (2006)
8. Chomicki, J., Godfery, P., Gryz, J., Liang, D.: Skyline with Presorting. In: ICDE (2003)
9. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with Presorting: Theory and Optimizations. In: Intelligent Information Systems (2005)
10. Fagin, R.: Combining Fuzzy Information from Multiple Systems. In: PODS, pp. 216–226 (1996)
11. Fagin, R., Lote, A., Naor, M.: Optimal Aggregation Algorithms for Middleware. In: PODS 2001 (2001)
12. Godfrey, P., Shipley, R., Gryz, J.: Maximal Vector Computation in Large Data Sets. In: VLDB (2005)

13. Guentzer, U., Balke, W., Kiessling, W.: Optimizing Multi-Feature Queries in Image Databases. In: VLDB 2000 (2000)
14. Guentzer, U., Balke, W., Kiessling, W.: Towards Efficient Multi-Feature Queries in Heterogeneous Environments. In: ITCC 2001 (2001)
15. Hwang, S., Chang, K.: Optimizing Top-k Queries for Middleware Access: A Unified Cost-based Approach. *ACM Trans. on Database Systems* (2007)
16. Ilyas, I.F., Beskales, G., Soliman, M.A.:
17. Kossmann, D., Ramsak, F., Rost, S.: Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In: VLDB (2002)
18. Lee, J., Hwang, S.: SkyTree: Scalable Skyline Computation for Sensor Data. In: SensorKDD (2009)
19. Lee, J., Hwang, S.: BSkyTree: Scalable Skyline Computation using Balanced Pivot Selection. In: EDBT (2010)
20. Lee, K.C., Zheng, B., Li, H., Lee, W.C.:
21. Lo, E., Yip, K.Y., Lin, K.I., Cheung, D.W.: Progressive Skylining over Web-Accessible Database. *Data & Knowledge Engineering* (2006)
22. Morse, M., Patel, J.M., Jagadish, H.:
23. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An Optimal and Progressive Algorithm for Skyline Queries. In: SIGMOD (2003)
24. Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T.: Access Path Selection in a Relational Database. In: SIGMOD 1979 (1979)
25. Zhang, S., Mamoulis, N., Cheung, D.W.: Scalable Skyline Computation Using Object-based Space Partitioning. In: SIGMOD (2009)

Chapter 4

Preference-Based Query Personalization

Georgia Koutrika, Evaggelia Pitoura, and Kostas Stefanidis

Abstract. In the context of database queries, computational methods for handling preferences can be broadly divided into two categories. Query personalization methods consider that user preferences are provided as a user profile separately from the query and dynamically determine how this profile will affect the query results. On the other hand, preferential query answering methods consider that preferences are explicitly expressed within queries. The focus of this chapter is on query personalization methods. We will first describe how preferences can be represented and stored in user profiles. Then, we will discuss how preferences are selected from a user profile and applied to a query.

4.1 Introduction

Preferences have a central role in many disciplines, including philosophy, decision theory and rational choice theory, a subject that in its turn permeates modern economics and other branches of formalized social science. How preferences are conceived and analyzed varies between these disciplines. For example, they could be conceived of as an individual's attitude towards a set of objects, typically reflected in an explicit decision-making process [39]. Alternatively, one could interpret the term "preference" to mean evaluative judgment, in the sense of liking or disliking an object (e.g., [47]), which is the most typical definition employed in psychology.

Georgia Koutrika
IBM Almaden Research Center, USA
e-mail: koutrika@stanford.edu

Evaggelia Pitoura
University of Ioannina, Greece
e-mail: pitoura@cs.uoi.gr

Kostas Stefanidis
Norwegian University of Science and Technology, Norway
e-mail: kstef@idi.ntnu.no

In other disciplines, particularly in economics, there is a strong tradition to equate preference with choice [26].

Computer science fields, such as artificial intelligence, human-computer interaction and databases, also deal with preferences. Here, explicit preference modeling provides a declarative way to choose among alternatives, whether these are solutions of a problem that an agent has to solve, user interaction elements or answers of database queries.

In databases, attention has been given on representations of preferences and on computational methods for handling preferences in the context of queries. Preference representations fall into two main categories. *Qualitative* preferences are specified using binary predicates that relatively compare tuples (e.g., [13, 33]). For example, a tuple a is preferred over tuples b and c . *Quantitative* preferences are expressed by assigning scores to particular tuples (e.g., [3]) or query elements that describe sets of tuples (e.g., [38]), where a score expresses degree of interest. A tuple a is preferred over tuple b if the score of a is higher than the score of b .

Computational methods for handling preferences in the context of database queries can be broadly divided into two categories. In *query personalization*, user preferences are provided in the form of a user profile. The goal is to find at any time which preferences should be taken into consideration in the context of a given query and modify the query results accordingly. On the other hand, *preferential query answering* considers that preferences are explicitly expressed within queries as soft conditions. If all preferences are equally important then the Pareto optimal set, i.e., all tuples which are not dominated by any other tuple, is computed (for skyline queries, refer to Chapters 2 and 3). If preferences are expressed using scores and there is an aggregating function for combining the partial scoring predicates, then the top- k results are computed (for top- k queries, refer to Chapter 3).

The focus of this chapter is on query personalization. We will first discuss how preferences can be represented and stored in user profiles (Section 4.2). Then, we will discuss how preferences are selected from a user profile and applied to a query (Section 4.3). As a running example, we consider a database that stores information about movies whose schema graph is described in Figure 4.1.

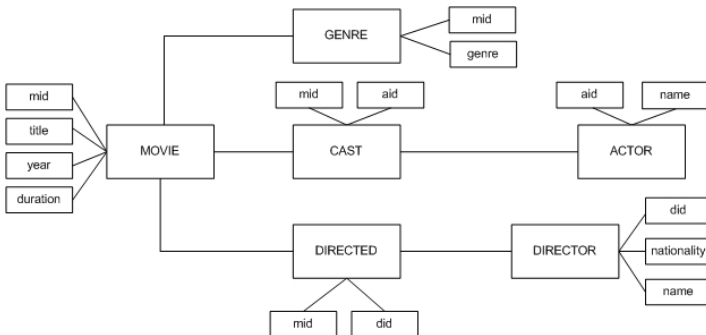


Fig. 4.1 An example database graph.

4.2 Preference Representation

Preferences express personal taste and are naturally context-dependent (or situated [31]), i.e., they may hold under specific conditions. For example, one may like watching comedies at home but prefer watching adventures at the movies. Or, one may like walking to work on a fine day but would always drive on rainy days. Or, one may always like driving to work under all weather conditions. Consequently, we consider that preferences are contextual and that, in their general form, have two parts, namely a context and a preference specification.

Definition 1 (Contextual Preference). A contextual preference CP is a pair (C, P) , where P describes the preference and C specifies the context, i.e., the conditions under which the preference holds.

A set of user preferences comprise a *user profile* U , which can be used for query personalization. We discuss how a user's preferences can be learnt in Section 4.4.

In the remaining section, first, we will elaborate on the specification of the context part C and then on the preference part P of a contextual preference. Then, we will show how individual preferences can be combined together and we will conclude the section with an example user profile.

4.2.1 Context Specification

The notion of context has been extensively studied in the context-aware computing community (e.g., [8], [43]), where different definitions of context have been proposed. A commonly used definition is the following [19].

“Context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and the application.”

Under this very general definition, user preferences can be also considered part of the user context, since they characterize the situation of a user. Here, we are interested in how the context determines when user preferences hold. In particular, we are interested in *external context*, that is a situation outside the database. Common types of external context include the *computing context* (e.g., network connectivity, nearby resources), the *user context* (e.g., accompanying people, location), the *physical context* (e.g., noise levels, temperature) and *time* [11].

A generic way to model external context is through a set of attributes, called *context parameters*. The set of context parameters used to model context depends on the specific application.

Definition 2 (External Context). Given a set of context parameters C_1, \dots, C_n with domains $dom(C_1), \dots, dom(C_n)$, respectively, an external context specification C is an n -tuple of the form (c_1, \dots, c_n) , where $c_i \in dom(C_i)$, $1 \leq i \leq n$.

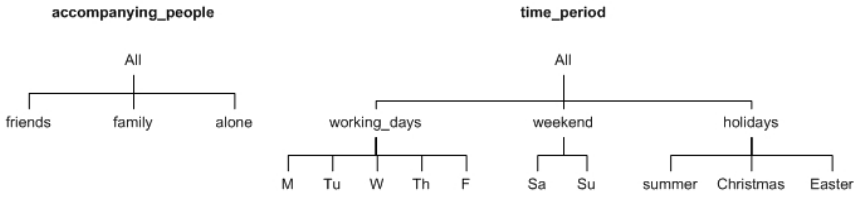


Fig. 4.2 Context hierarchies of *accompanying_people* and *time_period*.

The meaning of such a specification C in a contextual preference (C, P) is that the preference expressed by P holds only when the context parameters take the values in C . For example, for the movie database, we may consider two context parameters as relevant, namely, *accompanying_people* and *time_period*. In this case, a context specification $(family, Christmas)$ can be used to express a preference for specific movies in this context, that is, when the user is with her family during Christmas.

Additional flexibility in defining context specifications may be achieved by using context parameters that take values from hierarchical domains. By doing so, the specification of context C can be done at various levels of detail [42, 51, 52]. For instance, using a hierarchical context parameter to model time would allow us to express preferences that hold at the level of a day, a month or a year.

For example, for our movie database, consider that the two relevant context parameters, i.e., *accompanying_people* and *time_period*, take values from the hierarchical domains shown in Figure 4.2. Such hierarchies can be constructed using, for example, WordNet [43] or other ontologies (e.g., [27]).

The relationships between values at different levels of a hierarchy can be captured through a family of ancestor and descendant functions [55]. An ancestor function *anc* is responsible for assigning each value of a specific level to a value at a higher level, while a descendant function *desc* returns the descendants of a value at a specific level. For the hierarchies in Figure 4.2, *holidays* is an ancestor of *Christmas*, whereas the descendants of *weekend* are $\{Sa, Su\}$. Values at higher levels are considered more general than values in lower levels. The root of each hierarchy is the value *All* denoting the most general value for the corresponding parameter, while the values at the leaf nodes are the most specific or most detailed ones. In our example, value *holidays* is more general than value *Christmas*, while *family* and *friends* are at the same level of detail.

Example. Taking advantage of the hierarchical nature of context parameters, we can express preferences for movies (Figure 4.1) with different levels of details. For example, a user, say Julie, likes watching cartoons at weekends when accompanied by her family but enjoys watching action movies when alone. The context part of the first preference may be defined as $(family, weekends)$, while the second one as $(alone, All)$. We can also express preferences that hold independently of context. For example, say Julie enjoys also comedies at any time of the year and with any company. The context part for this preference can be expressed as (All, All) . \square

External context, called *situations*, is also used along with preferences in [31]. In this model, each context specification has an identifier *cid* and consists of a timestamp and location as well as other influences, such as physical state, current emotion, weather conditions and accompanying people. External contexts are uniquely linked through N:M relationships with preferences. Therefore, each contextual preference can be described as a (*cid*, *pid*) relationship instance, expressing that preference *pid* holds in context *cid*. Such descriptions may be saved in the database.

Finally, a knowledge-based approach is proposed in [9], where contextual preferences are called *preference rules*. In these rules, both the context and the preference specification parts are expressions in description logic.

4.2.2 Preference Specification

We consider that preferences are expressed for tuples of a relational database. Therefore, let us first introduce some related notation. A database comprises a set of relations. A relation R has a set \mathbf{A} of attributes. We will use RA_j to refer to an attribute in \mathbf{A} or simply A_j when R is understood. $dom(A_j)$ is the domain of values for attribute A_j . We use t to denote a tuple of R .

There are two fundamental ways of expressing preferences. In *qualitative* preference specification, preferences are defined between pairs of tuples. Formally, a qualitative preference P is a binary relation \succ_P over R , where for $t_i, t_j \in R$, $t_i \succ_P t_j$ means that t_i is preferred over t_j . We also say that t_i *dominates* t_j . In *quantitative* preference specification, preferences are expressed by assigning numerical scores to each tuple t in R indicating the user interest in t . In this chapter, we focus on quantitative preference specification.

It is typical to consider scores in the range of $[0, 1]$, where the closer the score to 1, the higher the user interest for the tuple. Clearly, one could consider different domains of values for scores. Intuitively, larger scores will show stronger preference. Ideally, there would be a perfect ranking function that would map each tuple $t \in R$ to a $score(t)$ [3].

Definition 3 (Ranking Function). Given a relation R and its attribute set \mathbf{A} , a ranking function $score(t) : dom(\mathbf{A}) \rightarrow [0, 1]$ maps tuples to scores.

The ranking function imposes an order on the tuples of R . In general, t_i in R is preferred over t_j in R , or t_i is ranked higher than t_j , for a ranking function $score$, denoted, $t_i \succ_{score} t_j$ if and only if, $score(t_i) > score(t_j)$. Note that the qualitative model is in general more expressive than the quantitative one. It has been shown that not all qualitative preferences can be captured by a single function unless they imply a weak order of the tuples [26].

If such an elaborate ranking function existed, it would yield a perfectly fine-grained ranking of tuples (e.g., movies, actors, and so forth) and would allow to return highly personalized answers to user queries. Unfortunately, such perfect ranking functions are also rather rare in practice. One reason is that user preferences are incomplete by nature. For instance, a user may not fully know or understand her

preferences on movies. Furthermore, individual user judgments may be conflicting and produce more than one ranking [15, 32]. For example, a user liked Million Dollar Baby, which is a movie directed by Clint Eastwood, but did not like Torino, another movie by the same director. What is her preference for Clint Eastwood, then?

Instead of assigning preference scores to individual tuples, a more convenient way to describe preferences is to assign preference scores to query constructs, such as selection and join conditions. These constructs can be then naturally used to personalize a query. Such scores can be assigned to selection conditions and join conditions.

Definition 4 (Tuple Preference). A preference P for a set of tuples of a relation R in the database is expressed as a pair $(condition, score)$ where $condition$ involves a set \mathbf{B} of attributes, $\mathbf{B} \subseteq \mathbf{A}$, and $score$ is a function: $dom(\mathbf{C}) \rightarrow [0, 1]$, where $\mathbf{C} \subseteq \mathbf{A}$ and \mathbf{A} is the set of all attributes of relation R .

The meaning is that all tuples from R that satisfy $condition$ are assigned a score through function $score$. Function $score$ may assign the same constant value to all tuples that satisfy the $condition$ or assign different scores to these tuples based on the values of the attributes in \mathbf{C} . Again, for a tuple preference P , a tuple t_i is preferred over a tuple t_j , if and only if, $score(t_i) > score(t_j)$. Consequently, each preference generates a partial ranking or order of the tuples in R .

Depending on $condition$, a preference can be one of the following types [37]:

- (i) *Selection preference.* If $condition$ is a conjunction of atomic selections involving a set of attributes and atomic joins transitively connecting these attributes to R on the database graph, then P is called a selection preference.
- (ii) *Join preference.* If $condition$ is a conjunction of atomic join conditions representing the transitive join of relations R and R' on the database graph, then P is called a join preference.

A selection preference indicates a user's preference for tuples from R that satisfy the transitive selections in $condition$. In their simplest form, selection preferences can assign a preference score to a set of tuples based solely on the values of the tuple [36, 38, 42, 52]. That is, both \mathbf{B} and \mathbf{C} include only attributes in R . These preferences are also called *intrinsic* [13]. A join preference indicates the degree in which preferences for tuples in R are influenced by preferences on joining tuples.

Example. To illustrate preferences, we consider the database shown in Figure 4.1. Julie's preferences for movies are captured in her profile. For instance, she likes comedies (P_1).

(P_1) GENRE: GENRE.genre = 'comedy'

0.85

She also likes family adventures with duration around one and a half hour (P_2). A function, such as $f_d(MOVIE.duration, 90min) = 0.9 * (1 - \frac{|MOVIE.duration - 90|}{90})$, can be used to give scores to family adventures based on their duration.

(P_2) MOVIE : MOVIE.mid = GENRE1.mid and GENRE1.genre = 'family' and
 MOVIE.mid = GENRE2.mid and
 GENRE2.genre = 'adventure' $f_d(\text{MOVIE.duration}, 90\text{min})$

Moreover, Julie considers the director of a movie to be more important than the genre of the movie (P_3 and P_4). This is expressed through preferences that use join conditions.

(P_3) MOVIE : MOVIE.mid = DIRECTED.mid and DIRECTED.did = DIRECTOR.did 1.0

(P_4) MOVIE : MOVIE.mid = GENRE.mid 0.7

One of her favorite directors is Alfred Hitchcock (P_5).

(P_5) DIRECTOR : DIRECTOR.name = 'A.Hitchcock' 0.7 □

In addition to preferences for tuples, we can also have preferences for attributes.

Definition 5 (Attribute Preference). A preference P for an attribute A_j of a relation R in the database is expressed as a pair (A_j, score) .

Attribute preferences can have different interpretations. One possible interpretation is to set priorities among tuple preferences based on the attributes involved in the preferences (e.g., [28]). Alternatively, attribute preferences can express priorities among the attributes to be displayed in the result of a query (e.g., π -preferences [42]).

Example. Julie considers the release year a movie more important than its duration (preferences P_6 and P_7).

(P_6) MOVIE : duration 0.7

(P_7) MOVIE : year 1.0

If we interpret P_6 and P_7 as priorities among attributes to display in the result, then the year of a movie is a more interesting piece of information to display than its duration. □

It is also possible to score attributes using automatic techniques that estimate the usefulness of an attribute and its visibility in the results of a query [18, 41].

One may consider the condition part of a preference to be a form of a context specification [2], in the sense that the preference holds only for the tuples that satisfy the conditions. For example, say Julie, wants the release year for animations to be after 2000, while for all other movies the year of release should be after 1960. Hence, her preference on more recent movies holds only for, or in the context of, animations. The context for this preference is described by the condition ($\text{MOVIE.mid} = \text{GENRE.mid}$ and $\text{GENRE.genre} = \text{'animation'}$). This type of context is dictated by the data items in the database and thus can be considered *internal* as opposed to external context that depends on conditions on items outside the database.

In general, such conditional preferences have received little attention in databases but they have been studied extensively in AI. A commonly used graphical notation for their representation is a *conditional preference network*, or *CP-net* (e.g., [71]).

CP-nets use conditional *ceteris paribus* (all else being equal) semantics. A CP-net over a set of attributes $\mathbf{A} = \{A_1, \dots, A_d\}$ is a directed graph in which there is a node for each attribute in \mathbf{A} . If an edge from an attribute A_j to an attribute A_i exists in the graph, then A_j is an ancestor of A_i . Let Z_i be the set of all ancestors of A_i . Semantically, the preferences over A_i depend on the attributes Z_i . Each attribute A_i is annotated with a *conditional preference table*, *CPT*, describing the preferences over A_i 's values given a combination of its ancestor values. That is, the *CPT* of A_i contains a set of preference expressions of the form $z_i : a_{i_1} \succ a_{i_2}$, $z_i \in \text{dom}(Z_i)$ and $a_{i_1}, a_{i_2} \in \text{dom}(A_i)$. This statement defines the conditional preference of a_{i_1} over a_{i_2} under context z_i . In particular, a tuple with a_{i_1} is preferred over a tuple with a_{i_2} when the value of Z_i is z_i , only if the values of the remaining attributes are equal.

There are also a few recent proposals of using CP-nets to represent database preferences. *Hierarchical CP-nets* introduced in [44] extend CP-nets by adding, among others, attribute preferences; a preference over an attribute is of higher priority than the preferences over the descendants of this attribute. Thus, each edge of the net expresses both the conditional dependence and the relative importance between different attributes. In *incomplete CP-nets*, preferences are only partially specified resulting in pairs of tuples being incomparable or indifferent in some contexts [14]. This approach proposes decoupling CP-nets from the *ceteris paribus* semantics used so far, and using the totalitarian semantics, that intuitively consider a tuple t_i to be preferred over t_j if the value of t_i at some attribute is preferred over the corresponding value of t_j and none of the other values of t_j is preferred over that of t_i (Pareto semantics). Finally, a new preference constructor to capture the *ceteris paribus* semantics is introduced in [22] in the context of translating a CP-net into an expression in the formal preference language over strict partial orders of [33].

4.2.3 Combining Preferences

Each tuple preference yields a partial ranking of the database tuples. Given a set of preferences defined over a relation, we can define different ways to combine the partial rankings into a total ranking of the tuples. In this section, we assume that the applicable preferences have the same external context, thus, in the following, we ignore the context part. We discuss when preferences are generally applicable to a query, taking into account context, in Section 4.3.1

We consider first the case in which a set of preferences may order the same pair of tuples differently. This situation is sometimes called a *conflict*.

Definition 6 (Conflicting Preferences). Two tuple preferences $P_x: (\text{condition}_x, \text{score}_x)$ and $P_y: (\text{condition}_y, \text{score}_y)$ defined over the same relation R are *conflicting* for a tuple t , if and only if, t satisfies both condition_x and condition_y and $\text{score}_x(t) \neq \text{score}_y(t)$.

Some conflicts can be resolved using *conflict resolution rules*. An intuitive such rule is *preference overriding* [37]. P_x is *overridden* by P_y , if condition_y is subsumed by condition_x , that is, for all potential tuples t in R , if t satisfies condition_y then it also

satisfies *condition_x*. Then, preference P_y is called *specific*, while P_x is called *generic* and is given priority over P_y , that is P_x is used, only when P_y does not apply. For example, P_x could be a preference for comedies and P_y be a preference for comedies with A. Sandler. Then, a comedy with A. Sandler is assigned $score_y(t)$, while all other comedies are assigned $score_x(t)$.

Conflicts can be generally resolved by using aggregating functions, which combine all partial scores into one.

Definition 7 (Aggregating Function). Let t be a tuple in a relation R that satisfies a set of preferences P_1, \dots, P_n defined over R and as a result receives the partial scores $score_{P_1}(t), \dots, score_{P_n}(t)$. The final score of t can be computed using an aggregating or combining function $F : [0, 1]^n \rightarrow [0, 1]$ that combines the partial scores into a single one.

Commonly used aggregating functions include weighted summation or average, “*min*” and “*max*”. In general, aggregating functions can be distinguished into [36]:

- (i) *Inflationary*: when the score of a tuple that satisfies multiple preferences together increases with the number of these preferences;
- (ii) *Dominant*: when the score assigned by one of the preferences dominates; and
- (iii) *Reserved*: when the combined score of each tuple lies between the highest and the lowest scores assigned to it.

Several combining functions have been defined in the context of multimedia databases in [23, 24]. There, each tuple is assigned a score $s \in [0, 1]$ based on how well it satisfies an atomic query. The objective is to compute an aggregated score for each tuple based on how well the tuple satisfies a boolean combination (disjunction or conjunction) of such atomic queries. Several interesting properties that aggregation function needs to satisfy are defined.

The problem of combining a set of partial rankings into a single ranking of items is also referred to as *rank aggregation*. Rank aggregation is a general problem. It has been studied in several fields including *social choice*, which studies the problem of determining the ranking of alternatives that is the most appropriate for a group given the individual opinions of its members [16, 54], *web meta-search*, which combines ranked lists of web pages produced by different search engines [15, 21], *database middleware*, where the problem is to combine multiple rankings of a set of objects based on preferences defined on different dimensions of these objects [25] and *collaborative filtering*, which combines known preferences of a group of users to predict preferences for unrated items [1].

Methods used by rank aggregation can be exploited to combine partial rankings that are produced by preferences. We outline two such methods from voting theory. One approach is the Borda count. Given a tuple t and n partial rankings, the proposed combining function assigns as a final score to t the sum of its positions in the n rankings [6]. Another approach is known as the Copeland index. In this case, the final score assigned to t is based on both the number of tuples that t dominates and the number of tuples t is dominated by in all the n rankings in which t participates.

So far we have seen methods for combining preferences (or their resulting partial rankings) over a single relation. It is possible to take into account preferences that are not explicitly defined over a relation but can implicitly produce a ranking of the tuples in the relation through the join preferences that relate this relation to neighboring relations on the database schema graph. Such preferences are called implicit [36].

Two preferences P_x and P_y are *composeable* [36], if and only if: (i) P_x is a join preference of the form $(condition_x, score_x)$ connecting R_x to a relation R_y , and (ii) P_y is a join or selection preference $(condition_y, score_y)$ on R_y .

Definition 8 (Implicit Preference). An *implicit preference* $(condition, score)$ is defined from the composeable preferences P_x and P_y , on $R \equiv R_x$, and: (i) *condition* is the conjunction of the conditions $condition_x, condition_y$ and (ii) *score* is a function of the degrees of interest of the two preferences, i.e., $score = f(score_x, score_y)$.

Several functions are possible for determining the score of an implicit preference. Non-increasing functions, such as the product and the minimum, are more natural since the resulting (implied) preference cannot be stronger than its supporting preferences.

Example. As we have seen in previous examples, Julie likes director A. Hitchcock (preference P_5) and she considers the director of a movie quite important (preference P_3). We can define an implicit preference for movies that are directed by A. Hitchcock (taking, for example, the product of the preference scores), as follows:

```
(P8) MOVIE :    MOVIE.mid = DIRECTED.mid and DIRECTED.did = DIRECTOR.did
                and DIRECTOR.name = 'A.Hitchcock'                                0.7
```

Finally, note that so far, we have considered preferences of the same user. In many applications, one needs to combine partial rankings that represent preferences of different people to produce a single ranking for all of them. This is often called *group preference aggregation*. Many approaches have been proposed [40]. For example, a *consensus function* that takes into account both the tuple scores for the users and the level at which users disagree with each other can be used [4]. Different strategies can be implemented, such as the least misery, where the idea is that a group is as happy as its least happy member.

4.2.4 Example: A User Profile

Above, we have illustrated separately how contexts and preferences can be represented. Figure 4.3 presents a set of contextual preferences.

In particular, given the database shown in Figure 4.1, CP_1 defines that Julie likes comedies when with friends during holidays, while, she likes adventures when with family during holidays (CP_2). When she is alone at weekends, her favorite director is Alfred Hitchcock (CP_3). Knowing the production year of a movie is always important (CP_4) but the movie's duration is less important when she is with friends than when she is alone (CP_5 and CP_6). Finally, with respect to (CP_7), (CP_8) and

(CP_9), she always considers both the director and the actors of a movie more important than the genre.

To map contextual preferences over a database, we extend the concept of the *personalization graph*, introduced in [38], to include context. More specifically, let $G(V, E)$ be a directed graph that is an extension of the database schema graph. Graph nodes correspond to schema relations, attributes and values for which a user has a preference. Edges are selection edges, representing a possible selection condition from an attribute to a value node, join edges, representing a join between relations, and projection edges connecting an attribute to its container relation. Selection preferences map to selection edges. Join edges capture join preferences. Attribute preferences map to the projection edges. All kinds of edges are associated with pairs of the form $(C, score)$, where C defines the context under which the preference that corresponds to the edge holds and $score$ stands for the preference score. Figure 4.4 illustrates a personalization graph that captures the preferences given in Figure 4.3.

(CP_1)	(C_1) (friends, holidays)	(P_1) GENRE :	GENRE.genre = 'comedy'	0.85
(CP_2)	(C_2) (family, holidays)	(P_2) GENRE :	GENRE.genre = 'adventure'	0.8
(CP_3)	(C_3) (alone, weekend)	(P_3) DIRECTOR :	DIRECTOR.name = 'A.Hitchcock'	0.9
(CP_4)	(C_4) (All, All)	(P_4) MOVIE :	MOVIE.year	1.0
(CP_5)	(C_5) (alone, All)	(P_5) MOVIE :	MOVIE.duration	1.0
(CP_6)	(C_6) (friends, All)	(P_6) MOVIE :	MOVIE.duration	0.7
(CP_7)	(C_7) (All, All)	(P_7) MOVIE :	MOVIE.mid = DIRECTED.mid and DIRECTED.did = DIRECTOR.did	1.0
(CP_8)	(C_8) (All, All)	(P_8) MOVIE :	MOVIE.mid = CAST.mid and CAST.aid = ACTOR.aid	1.0
(CP_9)	(C_9) (All, All)	(P_9) MOVIE :	MOVIE.mid = GENRE.mid	0.7

Fig. 4.3 Example contextual preferences.

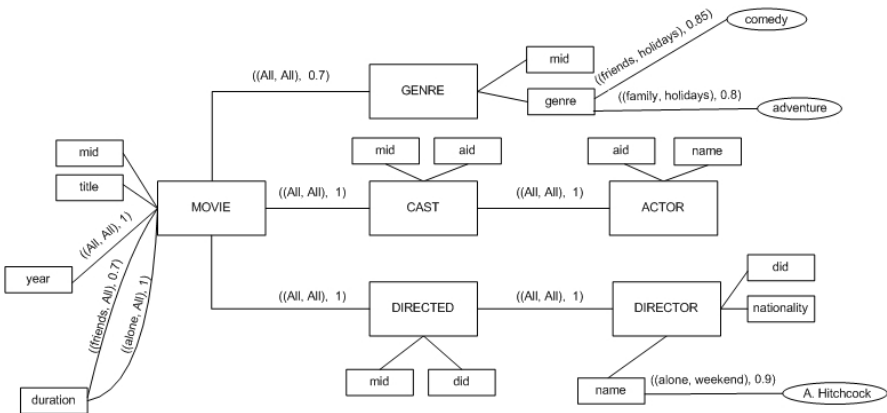


Fig. 4.4 Personalization graph enhanced with context.

In the sequel, we will use the terms contextual preferences and preferences interchangeably.

4.3 Personalizing Queries Using Preferences

Query personalization methods consider that user preferences are provided as a user profile. At query time, the goal is to find which preferences should be taken into consideration in the context of a query and integrate them into query processing.

At a high level, query personalization involves two phases (Figure 4.5): preference selection and personalized query processing. *Preference selection* determines which preferences from a user profile can possibly affect the query based on how they relate to the query and the external context at query time. For example, assume that Julie wants to find a comedy (the query) to watch with her family (the external context). Then, her preferences for dramas may not be applicable since she is searching for comedies. In addition, her movie preferences in the company of friends are outside the current context. *Personalized query processing* involves integrating preferences into the query and returning a preferential (or personalized) answer. Preferences may be used to rank the query results and potentially return to the user only the top-k most preferred ones.

In the following sections, we describe each of the two phases in detail.

4.3.1 Preference Selection

Given a query and a user profile, the first phase of query personalization identifies which preferences can be combined with the query. Let (C^Q, Q) be the query, which is also contextual, i.e., the query Q is formulated over the database and enhanced with a specification of context, denoted C^Q . The query context, C^Q , is described using the same set of context parameters used to specify preferences. C^Q may be postulated by the application or be explicitly provided by the users as part of their queries. Typically, in the first case, the query context corresponds to the current context, that is, the context surrounding the user at the time of the submission of her query. Such information may be captured using appropriate devices and mechanisms, such as temperature sensors or GPS-enabled devices for location. Besides this implicit context, users may explicitly specify C^Q . For example, Julie may express an exploratory query asking for interesting movies to watch with her *family* over the coming *weekend*.

Let (C^P, P) denote a preference from the user profile. We decide whether preference (C^P, P) can be used with query (C^Q, Q) based on:

- (i) *Context Matching*: how close their contexts C^P and C^Q are, and
- (ii) *Preference Relevance*: whether P is relevant to (a subset of) the results of Q .

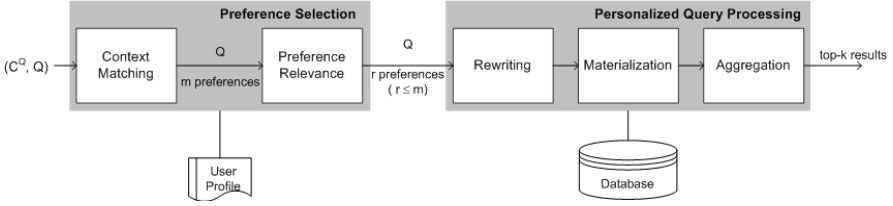


Fig. 4.5 Query Personalization Workflow.

We discuss context matching and preference relevance next. Then, we describe a methodology for selecting preferences from a user profile to be combined with a given user query.

Context Matching. Clearly, in terms of context, a preference (C^P, P) can be used to personalize a query (C^Q, Q) , if $C^P = C^Q$. However, when there are no such preferences in the profile, or when their number is small, we may need to select in addition preferences whose context C^P is not necessarily the same with C^Q but close enough to it. To determine how close contexts C^P and C^Q are, we rely on appropriate distance or similarity measures.

When context parameters take values from hierarchical domains, such as those in Figure 4.2, we can relate contexts expressed at different levels of detail. For instance, we can relate a context in which the parameter *time_period* is instantiated to a specific occasion (e.g., *Christmas*) with a context in which the same parameter describes a more general period (e.g., *holidays*). Intuitively, a preference defined for a more general value, e.g., *holidays*, may be considered applicable to a query about a more specific one, e.g., *Christmas*. In general, we can relate the external context of a preference (C^P, P) to the context C^Q of a query, if C^P is more general than C^Q , that is, if the context values specified in C^P are equal to or more general than the ones specified in C^Q . In this case, we say that C^P covers C^Q [52].

Given $C^P = (c_1^P, \dots, c_n^P)$ and $C^Q = (c_1^Q, \dots, c_n^Q)$, both defined with the help of a set of hierarchical parameters C_1, \dots, C_n , where C^P covers C^Q , one way to quantify their relevance is based on how far away are their values in the corresponding hierarchies.

$$dist_H(C^P, C^Q) = \sum_{i=1}^n d_H(level(c_i^P), level(c_i^Q)),$$

where $level(c_i^P)$ (resp. $level(c_i^Q)$) is the hierarchy level of value c_i^P (resp. c_i^Q) of parameter C_i and d_H is equal to the number of edges that connect $level(c_i^P)$ and $level(c_i^Q)$ in the hierarchy of C_i .

A similar metric is the *relevance index*, which is defined as follows [42]:

$$relevance(CP) = \frac{dist(C^Q, C_{root}^P) - dist(C^P, C^Q)}{dist(C^Q, C_{root}^P)},$$

where $\text{dist}(C^Q, C^P_{root})$ represents the highest possible distance of the query context with regard to any other context for which a preference exists. Preferences whose context is the query context have the maximum relevance index, that is 1, while preferences whose context is the most general one based on the available hierarchies have the minimum relevance, that is 0.

Generalizing the discussion above, it is also possible to relate a context C^P to a context C^Q , if C^P can be attained by relaxing zero or more parameters of C^Q in any of the following ways: a context parameter may be relaxed *upwards* by replacing its value by a more general one, *downwards* by replacing its value by a set of more specific ones or *sideways* by replacing its value by sibling values in the hierarchy. Given all these possible relaxations, appropriate distance metrics that exploit the number of relaxed parameters and the associated depth of such relaxations can be employed to measure how well context C^P matches C^Q [53].

Preference Relevance. Intuitively, the preference part P of a preference (C^P, P) is relevant to (a subset of) the results of Q , if combined together they yield an interesting, non-empty result. In practice, however, it may be difficult to decide when a preference is relevant to a query.

Consider, for example, a preference P on the genre of a movie and a query Q about actors born before 1970. The preference seems irrelevant to the query since it refers to a different “concept” (i.e., movies) from the concepts of the query (i.e., actors). On the other hand, one could consider the preference to be implicitly related to the query, since actors are connected to movies. Perhaps P could be used to rank the actors in the result of Q based on the genre of the movies in which each actor appears. Consequently, all preferences that are explicitly connected to the query as well as those implicitly connected to it can potentially be relevant [38].

A special form of relevance is applicability. A preference P is *applicable* to a query Q , if the execution of Q combined conjunctively with P (over the current database instance) yields a non empty result. This may be also called *instance applicability*. For example, consider a query about recent movies and a preference for movies directed by Steven Spielberg. This preference is instance applicable only if the database contains recent entries of this director. In general, instance applicability can only be checked by actually executing the query with the preference.

Another type of applicability is semantic applicability. A preference P is not *semantically applicable* to a query Q , if the execution of Q combined conjunctively with P over any database instance yields an empty result. To decide whether a preference is semantically applicable to a query, knowledge outside the database may be needed. Consider as an example a query about comedies. Then, a preference for movies directed by Andrei Tarkovsky is not semantically applicable to this query, since this director has not directed any comedies.

In some special cases, the applicability of a preference P to a query Q can be determined simply by a *syntactic analysis* of P and Q . For example, a query about movies released after 2000 and a preference for movies released prior to 1990 are conflicting and will return an empty result when combined through a conjunction. On the other hand, a preference for movies with actor Ben Stiller is syntactically

applicable to a query for movies with Julia Roberts since it is possible that the two actors play in the same movie. Note that a preference P that is syntactically applicable to Q , it is not necessarily instance applicable. However, the reverse always hold.

Other definitions of preference relevance have also been considered. For example, in [9], both contexts and preferences are defined through description logics concept expressions. Contextual preferences are considered relevant to a query if their contexts are the same with or more general than the query context and their preferences contain concepts which can be mapped to certain relations of the query. In [50], the focus is on personalization of keyword queries, where contexts, preferences and queries are specified through keywords. A contextual preference is considered relevant to a query if its context is the same with or more general than the query, i.e., contains a subset of the query keywords.

4.3.1.1 Selecting Preferences Based on Their Context and Relevance

Given a query and a user profile, the first phase of query personalization, i.e., preference selection, identifies preferences that can be combined with the query. This phase can be conceptually divided into two steps: context matching and selection of relevant preferences. The first step identifies the preferences from the user profile with contexts closest to the query context. Among those preferences, the ones that are relevant to the query are selected in the second step.

Selecting Preferences based on Context. Given a query context $C^Q = (c_1^Q, \dots, c_n^Q)$, $c_i^Q \in \text{dom}(C_i)$, $1 \leq i \leq n$, and a user profile U , the first step of preference selection identifies the m preferences in U with contexts closest to the query context. For this purpose, a special data structure, called *profile tree* can be used [52]. A profile tree offers a space-efficient representation of contexts defined in the user profile U by taking advantage of the co-occurrences of context values in U .

Given a set U of external contextual preferences defined over n context parameters C_1, \dots, C_n , the profile tree for U has $n+1$ levels. Each one of the first n levels corresponds to one of the parameters. For simplicity, assume that parameter C_i is mapped to level i of the tree. At the first level of the tree, there is a single root node. Each non-leaf node at level l , $1 \leq l \leq n$, contains a set of cells of the form $[key, pt]$, where $key \in \text{dom}(C_l)$ and appears in some context of a contextual preference in U . No two cells within the same node contain the same *key* value. The pointer *pt* of each cell of the nodes at level l , $1 \leq l < n$, points to the node at the next lower level (level $l + 1$) containing all the distinct values of the parameter C_{l+1} that appeared in the same context with *key*. The pointers of cells of the nodes at level n point to leaf nodes, where each one contains the preference part of the contextual preferences with context that corresponds to the path leading to it. As a concrete example, consider the profile tree of Figure 4.6 constructed for the contextual preferences of Figure 4.3.

Given the profile tree of U , the algorithm for selecting relevant preferences proceeds to locate the contexts that appear in at least one preference in U and cover

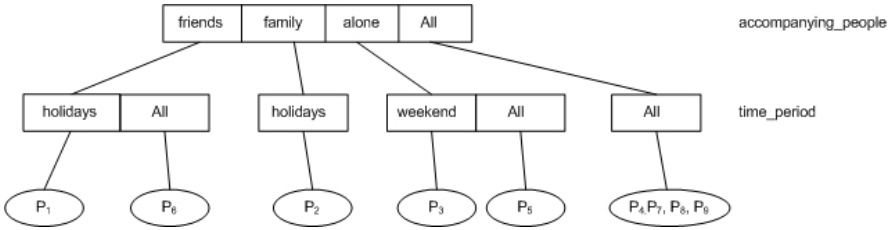


Fig. 4.6 An instance of a profile tree.

\mathcal{C}^Q through a top-down breadth-first traversal of the tree starting from its root. At each level i , all paths of length i whose context is either the same or more general than the prefix (c_1^Q, \dots, c_i^Q) of the input context are kept. The algorithm returns as its result the preferences of the leaf nodes at level $n + 1$ and the distances of the corresponding contexts from \mathcal{C}^Q .

To find the top- m preferences for \mathcal{C}^Q , we can sort the returned preferences for \mathcal{C}^Q on the basis of the distance of their contexts from \mathcal{C}^Q and select the m with the minimum distance. What is the right value for m ? By over-relaxing the query context, we may end up considering preferences that are unexpected for the given query context. On the other hand, by being very strict, we may end up with having very few preferences to consider for query personalization. Ideally, we would like to choose a value for m that would subsequently allow selecting at least top- r relevant preferences that can be applied to the given query specification, as we will see next. One way to achieve this is to interleave the two steps of preference selection. First, select a small number of preferences based on context. If among them there are fewer than r preferences that are relevant to the query, then further relax the query context in order to get the next preferences in order of the distance of their contexts from \mathcal{C}^Q .

Selecting Relevant Preferences. The output of the first step of preference selection is a set of preferences whose context match the query context. The second step determines which of these preferences are relevant and will be ultimately combined with the query.

All preferences that match the query context and are relevant to the query may be used for ranking and selecting the most interesting tuples returned by the query. Alternatively, preferences can be ranked based on their preference score and the top r most important ones can be selected for personalizing the query.

To select the top- r preferences, a preference selection algorithm may start from the preferences that match the query context and are relevant to the query. As long as fewer than r preferences are considered (and there are still candidate preferences), an approach is to iteratively consider additional preferences that are composable with those already known, so as to derive implicit ones that are also relevant to the query based on their syntactic characteristics. The set of preferences that are related to the query is kept ordered in decreasing preference score. When r preferences have

been found, the algorithm guarantees that these are the top r preferences related to the query [38].

Selecting the right value for r is an important factor of the success of personalization. Selecting too many preferences may lead to over-restricting the result, while selecting too few preferences may not suffice to express the initial user intent. It is possible that the user specifies r . A different approach is to view preference selection as an optimization problem with constraints. The parameters of the problem are the execution cost of the query, the size of the query result and the preference scores of the tuples in the query result. The objective is to select a set of relevant preferences that, in conjunction with the query, will optimize one of the parameters and satisfy constraints on the others [35].

4.3.2 Personalized Query Processing

The second phase of query personalization integrates preferences with query processing. Preferences may be used to rank the results of the query in order to select the top- k ones or they may impose additional (soft) constraints that combined with the query constraints will return results that satisfy some preferences.

Conceptually, personalized query processing proceeds in three steps:

- (i) *Rewriting*: A set of queries is built, each one describing a subset of the query results that satisfies one or more preferences. Each sub-query is built by extending the initial query by an appropriate qualification involving the participating preferences and assigns the corresponding preference score to each tuple in its results.
- (ii) *Materialization*: The new queries are executed and generate partial results ranked according to the user preferences. The initial query may be also executed if the output of query personalization will be just a ranking of the initial query results.
- (iii) *Aggregation*: The partial results are combined so that a single ranked list of results is output, which may contain the top- k items or items that satisfy some preferences.

Example. Consider the following query Q under some context C :

```
Q:  select MV.title from MOVIE MV, CAST CA, ACTOR AC
     where MV.mid = CA.mid and CA.aid = AC.aid and AC.name = 'S. Bullock'
```

Assume also that for the given query context, the top-3 relevant preferences (i.e., $r = 3$) are the following:

MOVIE :	MOVIE.year > 2005	1.0
MOVIE :	MOVIE.mid = GENRE.mid and GENRE.genre = 'comedy'	0.85
MOVIE :	MOVIE.duration < 110	0.7

Each one of the following queries captures one of these preferences and when executed will generate a list of results ranked accordingly:

Q_1 : `select distinct MV.title ,1.0 score
from MOVIE MV, CAST CA, ACTOR AC
where MV.mid = CA.mid and CA.aid = AC.aid and AC.name = 'S. Bullock' and
MV.year > 2005`

Q_2 : `select distinct MV.title ,0.85 score
from MOVIE MV, CAST CA, ACTOR AC, GENRE GE
where MV.mid = CA.mid and CA.aid = AC.aid and AC.name = 'S. Bullock' and
MV.mid = GE.mid and GE.genre = 'comedy'`

Q_3 : `select distinct MV.title ,0.7 score
from MOVIE MV, CAST CA, ACTOR AC
where MV.mid = CA.mid and CA.aid = AC.aid and AC.name = 'S. Bullock' and
MV.duration < 110` □

One approach to implementing the *materialization* and *aggregation* steps is to build a personalized query as the union of the sub-queries that map to the individual user preferences [38]. For example, we could build the following query:

Q' : `select distinct MV.title ,F(score) score
from Q1 Union All Q2 Union All Q3 group by MV.title order by F(score)`

An aggregating function F (Section 4.2.3) is used to combine partial scores for each tuple.

Given that each subquery produces a partial ranking of the query results, to construct a total ranking, instead of following the naive approach of computing the aggregate score of each tuple and ranking the tuples based on these scores, several more efficient algorithms have been proposed that can generate top- k results from the partial rankings.

A fundamental algorithm for retrieving the top- k tuples is the *FA* algorithm [25]. This algorithm considers two types of available tuple accesses: the *sorted* access and the *random* access. Sorted access enables tuple retrieval in a descending order of their scores, while random access enables retrieving the score of a specific tuple in one access. The main steps of the *FA* algorithm are the following:

1. First, do sorted access to each ranking until there is a set of k tuples, such that each of these tuples has been seen in each of the rankings.
2. Then, for each tuple that has been seen, do random accesses to retrieve the missing scores.
3. Compute the aggregate score of each tuple that has been seen.
4. Finally, rank the tuples based on their aggregate scores and select the top- k ones.

FA is correct when the aggregate tuple scores are obtained by combining their individual scores using a monotone function. This also holds for the *TA* algorithm [25]. *TA* ensures further that its stopping condition always occurs at least as early as the stopping condition of *FA*. Its main steps of *TA* are the following:

1. First, do sorted access to each ranking. For each tuple seen, do random accesses to the other rankings to retrieve the missing tuple scores.
2. Then, compute the aggregate score of each tuple that has been seen. Rank the tuples based on their aggregate scores and select the top- k ones.
3. Stop to do sorted accesses when the aggregate scores of the k tuples are at least equal to a threshold value that is defined as the aggregate score of the scores of the last tuples seen in each ranking.

Algorithms equivalent to TA have been proposed focusing on top- k results (e.g., [45, 29]). The PPA algorithm builds on the same idea to return personalized top- k results that satisfy a minimum number of preferences [36]. Instead of executing all sub-queries at once, the algorithm executes them in order of increasing selectivity. For each tuple returned by a sub-query Q_i , PPA executes a parameterized query that checks what other preferences are satisfied by the tuple apart from those covered by Q_i . The algorithm stops when the aggregate scores of the k tuples are at least equal to a threshold value or when the remaining not-executed queries Q_i cover fewer preferences than the number of preferences required to be satisfied in the output.

Furthermore, preferences may not be independent. For example, a preference for comedies can be combined with a preference for actor Adam Sandler but they are both overridden by a preference for ‘comedies with Adam Sandler’. Being able to detect such preference relationships can further help save execution time. The *Replicate_Diffuse* algorithm decides which preferences to process and in what order based on the preference relationships resulting in reduced preference processing [37]. The algorithm organizes the queries into a hierarchy, where at the higher level reside queries that integrate the most generic preferences (e.g., a preference for comedies), and as one moves down in the hierarchy, queries that map to more fine-grained preferences (e.g., a preference for ‘comedies with Adam Sandler’) are found. The algorithm executes the root queries and finds all tuples that satisfy at least one generic preference. For each tuple t and for each preference P satisfied by t , the algorithm tries to find (if exists) the most specific preference P' in the hierarchy that can override P and that is satisfied by t . In this case, the preference score of P' is taken into account when computing the total score for the tuple instead of the score of P .

A similar approach is found in [28], where given a set of preferences defined over a relation R the objective is to produce a ranking of R 's tuples that respects these preferences. To solve this problem, a query lattice with one node for each combination of values of different attributes appearing in the preferences is built. For example, for preferences: (i) A. Hitchcock is preferred over M. Curtiz or S. Spielberg, (ii) horror movies are preferred over drama movies and (iii) the director of a movie is as important as its genre, the query lattice of Figure 4.7 is constructed. A query is formulated for each node in the lattice. All queries in a specific block produce equally preferable results. The queries of each block are successively executed starting from the queries of the top block and going down the lattice.

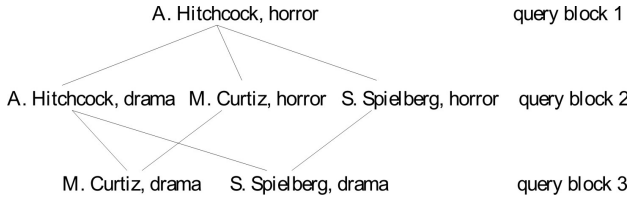


Fig. 4.7 Query lattice example.

4.4 Preference Learning

In this section, we discuss how preferences can be learnt and stored in a user profile.

Learning and predicting preferences in an automatic way has attracted much current attention in the areas of machine learning, knowledge discovery and artificial intelligence. Approaches to preference learning can be classified along various dimensions. Depending on *the model* learned, we may distinguish between learning pairwise orderings of items (i.e., qualitative preferences) and learning a utility function (i.e., quantitative preferences). Depending on *the type of information provided as input*, the learning algorithm may or may not use as input positive and/or negative examples, in analogy to supervised and unsupervised learning, and may or may not use relevance feedback from the users. Yet another distinction is on whether the input data is in the form of pairwise preferences, that is, the input is an order relation, or the input data is a set of items for which some knowledge about the associated preference degree is available. Another dimension for differentiating preference learning is based on *the data mining task* they use, such as associative rule mining, clustering or classification. Finally, the application also determines the preference learning process, since it affects both the form of input data (for example, clickthrough data or user ratings) as well as the desired use of the learned preferences (for example, personalized search results or recommendations).

Most methods for preference learning utilize information of the past user interactions in the form of a history or log of transactions. User clickthrough data, namely the query-logs of a search engine along with the log of links that the user actually clicked on from those in the presented ranked list is the input used by the method presented in [32]. The fact that a user clicked on a link l_i and did not click on a link l_j ranked higher in the list than l_i , is interpreted as a user preference of l_i over l_j . Clickthrough data are used as training data to learn a ranking function that, for each query q , produces an order of links based on their relevance to q . This is achieved by selecting the function that produces the order having the minimal distance from the orders inferred from the clickthrough data. A support vector machine (SVM) algorithm is used.

User logs in the form of relational instances are used as input in [30]. Since there is no explicit ranking information in the log of relations, to detect preferences, the frequencies of the different attribute values, i.e., their number of entries, in the log relation are used. Then, x is preferred over y , if and only if, $freq(x) > freq(y)$.

Preferences between values of individual attributes are used to infer various types of preferences as defined in [33].

User feedback is used for improving preference learning. For example, in [15], the authors consider the problem of learning how to rank items given the feedback that an item should be ranked higher than another. For a set of items \mathcal{I} , the employed preference function $Pref(i_1, i_2)$, $Pref: \mathcal{I} \times \mathcal{I} \rightarrow [0, 1]$, returns a value indicating which item, i_1 or i_2 , is ranked higher. The learning phase of such a function takes place in a sequence of rounds. At each round, items are ranked with respect to $Pref$. Then, the learner receives feedback from the environment. The feedback is assumed to be an arbitrary set of rules of the form “ i_1 should be preferred to i_2 ”. Given that $Pref$ is a linear combination of n primitive functions, i.e., $Pref(i_1, i_2) = \sum_{j=1}^n w_j F_j(i_1, i_2)$, at each round the weights w_j are updated with regards to the *loss* of a function F with respect to the user feedback, where *loss* is the normalized sum of disagreements between function and feedback.

Moreover, applying machine learning techniques for learning ranking functions has recently attracted much attention in the research literature (e.g., [10, 56, 57]).

4.5 Conclusion and Open Issues

In this chapter, we focused on query personalization methods. We discussed how preferences are represented to express the user interest in specific items under different contexts. Then, given a user query and a set of user preferences, the appropriate preferences are selected and applied to the query. Preference selection is divided into (i) selection based on context matching and (ii) selection based on preference relevance. We examined different methods for integrating preferences into query processing.

Although our primary focus is on personalized query processing, it is worth noting that there are other potential applications of preferences. For example, preferences can be used for managing the contents of a cache [12] or for combining quality metrics, such as *quality of service*, i.e., query response time, and *quality of data*, i.e., freshness of data in the query answer [46]. Preferences are also applied to e-business applications [34] and in publish/subscribe systems [20].

There are still many open problems with regards to personalization. The following is a list of research directions for future work that we consider particularly promising:

- (i) *Hybrid Preference Models*: Most current approaches to representing preferences are either purely qualitative or purely quantitative. However, in real life, preferences may be both absolute (e.g., ‘I like comedies a lot’) or relative (e.g., ‘I like comedies better than dramas’). Instead of converting them into a single form, a hybrid preference model that allows capturing both qualitative and quantitative preferences would be very useful. Such a model would also bring several query processing challenges, such as how to rank the query results

when both relative (i.e., qualitative) and absolute (i.e. quantitative) preferences apply.

- (ii) *Probabilistic Preferences*. Recently, there has been considerable interest in processing uncertain, or probabilistic, data (e.g., [5, 17]). In probabilistic databases, tuples are associated with membership probabilities that define the belief that they should belong to the database. There has been also related work on processing top- k queries over uncertain data [49, 53], where both preference scores and probabilities of tuples are taken into account. On the other hand, preferences themselves involve some degree of uncertainty. For example, we may be certain that a user likes comedies when explicitly stated but we may be less certain when implying user preferences. Consequently, designing appropriate probabilistic preference models and extending processing methods to conform with them are interesting directions.
- (iii) *Efficient Processing of Preference Queries*. There are many open problems with regards to query personalization. The problem of selecting the appropriate preferences for personalizing a query is challenging, since there can be no single best solution. Furthermore, the performance issues of integrating preferences with query processing are still open. Whereas there has been work on specific preference operators, such as top- k , skyline and their variants, there is very little work on tightly integrating more general preference models within the database engine.
- (iv) *Group Preferences*. Finally, although there has been considerable work on dealing with preferences at the level of an individual person searching a database, there has been less work on processing group preferences, i.e., preferences of a group of people. To this direction, there are several interesting questions, such as solving preference conflicts among the members of the group and generating efficiently non-empty results that satisfy the group.

References

1. Adomavicius, G., Tuzhilin, A.: Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Trans. Knowl. Data Eng.* 17(6), 734–749 (2005)
2. Agrawal, R., Rantau, R., Terzi, E.: Context-Sensitive Ranking. In: *SIGMOD*, pp. 383–394 (2006)
3. Agrawal, R., Wimmers, E.L.: A Framework for Expressing and Combining Preferences. In: *SIGMOD*, pp. 297–306 (2000)
4. Amer-Yahia, S., Roy, S.B., Chawla, A., Das, G., Yu, C.: Group Recommendation: Semantics and Efficiency. *PVLDB* 2(1), 754–765 (2009)
5. Benjelloun, O., Sarma, A.D., Halevy, A.Y., Widom, J.: ULDBs: Databases with Uncertainty and Lineage. In: *VLDB*, pp. 953–964 (2006)
6. Borda, J.C.: Mémoire sur les Élections au Scrutin. *Histoire de l'Académie Royale des Sciences* (1781)

7. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., Poole, D.: CP-nets: A Tool for Representing and Reasoning with Conditional Ceteris Paribus Preference Statements. *J. Artif. Intell. Res.* 21, 135–191 (2004)
8. Brown, P.J., Bovey, J.D., Chen, X.: Context-Aware Applications: From the Laboratory to the Marketplace. *IEEE Personal Communications* 4(5), 58–64 (1997)
9. van Bunnigen, A.H., Feng, L., Apers, P.M.G.: A Context-Aware Preference Model for Database Querying in an Ambient Intelligent Environment. In: Bressan, S., Küng, J., Wagner, R. (eds.) *DEXA 2006*. LNCS, vol. 4080, pp. 33–43. Springer, Heidelberg (2006)
10. Burges, C.J.C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G.N.: Learning to Rank using Gradient Descent. In: *ICML*, pp. 89–96 (2005)
11. Chen, G., Kotz, D.: A Survey of Context-Aware Mobile Computing Research. Tech. Rep. TR2000-381, Dartmouth College, Computer Science (2000), <ftp://ftp.cs.dartmouth.edu/TR/TR2000-381.ps.z>
12. Cherniack, M., Galvez, E.F., Franklin, M.J., Zdonik, S.B.: Profile-Driven Cache Management. In: *ICDE*, pp. 645–656 (2003)
13. Chomicki, J.: Querying with Intrinsic Preferences. In: Jensen, C.S., Jeffery, K., Pokorný, J., Saltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) *EDBT 2002*. LNCS, vol. 2287, pp. 34–51. Springer, Heidelberg (2002)
14. Ciaccia, P.: Querying databases with incomplete CP-nets. In: *M-Pref* (2007)
15. Cohen, W.W., Schapire, R.E., Singer, Y.: Learning to Order Things. *J. Artif. Intell. Res. (JAIR)* 10, 243–270 (1999)
16. Condorcet, J.A.N.: *Éssai sur l' application de l' analyse á la Probabilité des Décisions Rendues á la Pluralité des Voix*. Kessinger Publishing (1785)
17. Dalvi, N.N., Suciu, D.: Efficient Query Evaluation on Probabilistic Databases. *VLDB J.* 16(4), 523–544 (2007)
18. Das, G., Hristidis, V., Kapoor, N., Sudarshan, S.: Ordering the Attributes of Query Results. In: *SIGMOD*, pp. 395–406 (2006)
19. Dey, A.K.: Understanding and Using Context. *Personal Ubiquitous Comput.* 5(1), 4–7 (2001)
20. Drosou, M., Stefanidis, K., Pitoura, E.: Preference-aware Publish/Subscribe Delivery with Diversity. In: *DEBS*, pp. 1–12 (2009)
21. Dwork, C., Kumar, R., Naor, M., Sivakumar, D.: Rank Aggregation Methods for the Web. In: *WWW10* (2001)
22. Endres, M., Kießling, W.: Transformation of TCP-net Queries into Preference Database Queries. In: *M-Pref* (2006)
23. Fagin, R.: Combining Fuzzy Information from Multiple Systems. In: *PODS*, pp. 216–226 (1996)
24. Fagin, R.: Fuzzy Queries in Multimedia Database Systems. In: *PODS*, pp. 1–10 (1998)
25. Fagin, R., Lotem, A., Naor, M.: Optimal Aggregation Algorithms for Middleware. In: *PODS* (2001)
26. Fishburn, P.C.: Preference Structures and Their Numerical Representations. *Theoretical Computer Science* 217(2), 359–383 (1999)
27. Fontoura, M., Josifovski, V., Kumar, R., Olston, C., Tomkins, A., Vassilvitskii, S.: Relaxation in Text Search using Taxonomies. *PVLDB* 1(1), 672–683 (2008)
28. Georgiadis, P., Kapantaidakis, I., Christophides, V., Nguer, E.M., Spyrtatos, N.: Efficient Rewriting Algorithms for Preference Queries. In: *ICDE*, pp. 1101–1110 (2008)
29. Güntzer, U., Balke, W.T., Kießling, W.: Optimizing Multi-Feature Queries for Image Databases. In: *VLDB*, pp. 419–428 (2000)

30. Holland, S., Ester, M., Kießling, W.: Preference Mining: A Novel Approach on Mining User Preferences for Personalized Applications. In: Lavrac, N., Gamberger, D., Todorovski, L., Blockeel, H. (eds.) PKDD 2003. LNCS (LNAI), vol. 2838, pp. 204–216. Springer, Heidelberg (2003)
31. Holland, S., Kießling, W.: Situated Preferences and Preference Repositories for Personalized Database Applications. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 511–523. Springer, Heidelberg (2004)
32. Joachims, T.: Optimizing Search Engines using Clickthrough Data. In: KDD, pp. 133–142 (2002)
33. Kießling, W.: Foundations of Preferences in Database Systems. In: VLDB, pp. 311–322 (2002)
34. Kießling, W., Fischer, S., Döring, S.: COSIMAB2B - Sales Automation for E-Procurement. In: CEC, pp. 59–68 (2004)
35. Koutrika, G., Ioannidis, Y.: Constrained Optimalities in Query Personalization. In: SIGMOD, pp. 73–84 (2005)
36. Koutrika, G., Ioannidis, Y.: Personalized Queries under a Generalized Preference Model. In: ICDE, pp. 841–852 (2005)
37. Koutrika, G., Ioannidis, Y.: Personalizing Queries based on Networks of Composite Preferences. *ACM Trans. Database Syst.* 35(2) (2010)
38. Koutrika, G., Ioannidis, Y.E.: Personalization of Queries in Database Systems. In: ICDE, pp. 597–608 (2004)
39. Lichtenstein, S., Slovic, P.: *The Construction of Preference*. Cambridge University Press, New York (2006)
40. Masthoff, J.: Group Modeling: Selecting a Sequence of Television Items to Suit a Group of Viewers. *User Modeling and User-Adapted Interaction* 14(1), 37–85 (2004)
41. Miah, M., Das, G., Hristidis, V., Mannila, H.: Standing Out in a Crowd: Selecting Attributes for Maximum Visibility. In: ICDE, pp. 356–365 (2008)
42. Miele, A., Quintarelli, E., Tanca, L.: A Methodology for Preference-based Personalization of Contextual Data. In: EDBT, pp. 287–298 (2009)
43. Miller, G.A.: WordNet: a Lexical Database for English. *Commun. ACM* 38(11), 39–41 (1995)
44. Mindolin, D., Chomicki, J.: Hierarchical CP-networks. In: M-Pref (2007)
45. Nepal, S., Ramakrishna, M.V.: Query Processing Issues in Image (Multimedia) Databases. In: ICDE, pp. 22–29 (1999)
46. Qu, H., Labrinidis, A.: Preference-Aware Query and Update Scheduling in Web-databases. In: ICDE, pp. 356–365 (2007)
47. Scherer, K.: What are Emotions? And how can they be Measured? *Social Science Information* 44, 695–729 (2005)
48. Schmidt, A., Aidoo, K.A., Takaluoma, A., Tuomela, U., Van Laerhoven, K., Van de Velde, W.: Advanced Interaction in Context. In: Gellersen, H.-W. (ed.) HUC 1999. LNCS, vol. 1707, pp. 89–101. Springer, Heidelberg (1999)
49. Soliman, M.A., Ilyas, I.F., Chang, K.C.C.: Top-k query processing in uncertain databases. In: ICDE, pp. 896–905 (2007)
50. Stefanidis, K., Drosou, M., Pitoura, E.: PerK: Personalized Keyword Search in Relational Databases through Preferences. In: EDBT, pp. 585–596 (2010)
51. Stefanidis, K., Pitoura, E., Vassiliadis, P.: Modeling and Storing Context-Aware Preferences. In: Manolopoulos, Y., Pokorný, J., Sellis, T.K. (eds.) ADBIS 2006. LNCS, vol. 4152, pp. 124–140. Springer, Heidelberg (2006)

52. Stefanidis, K., Pitoura, E., Vassiliadis, P.: Adding Context to Preferences. In: ICDE, pp. 846–855 (2007)
53. Stefanidis, K., Pitoura, E., Vassiliadis, P.: On Relaxing Contextual Preference Queries. In: MDM, pp. 289–293 (2007)
54. Taylor, A.: Mathematics and Politics: Strategy, Voting, Power and Proof. Springer, New York (1995)
55. Vassiliadis, P., Skiadopoulos, S.: Modelling and Optimisation Issues for Multidimensional Databases. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 482–497. Springer, Heidelberg (2000)
56. Zha, H., Zheng, Z., Fu, H., Sun, G.: Incorporating Query Difference for Learning Retrieval Functions in World Wide Web Search. In: CIKM, pp. 307–316 (2006)
57. Zhai, C., Lafferty, J.D.: A Risk Minimization Framework for Information Retrieval. Information Processing and Management 42(1), 31–55 (2006)
58. Zhang, X., Chomicki, J.: Semantics and Evaluation of Top- k Queries in Probabilistic Databases. Distributed and Parallel Databases 26(1), 67–126 (2009)

Chapter 5

Approximate Queries for Spatial Data

Alberto Belussi, Barbara Catania, and Sara Migliorini

Abstract. Approximation techniques for spatial data traditionally concern data capture and data representation issues. On the other hand, more recently developed approximation techniques refer to the query to be executed and not to data representation as in the the past monolithic Geographic Information Systems and for this reason they are called *query-based* approximation techniques. The aim of this chapter is to survey such approximation techniques and to identify the issues that from our point of view have still to be investigated to complete the picture. In particular, we observe that most of the proposed approaches for spatial approximate queries rely on the usage of quantitative, i.e., metric (distance-based), information. On the other hand, only few of them take into account qualitative information, e.g., topological and cardinal spatial relations. Based on this consideration, we provide new types of queries relying on qualitative relations and we discuss how the query processing algorithms already defined for metric relations can be extended to cope with qualitative information.

5.1 Introduction

With respect to traditional, non geo-referenced data, spatial data are characterized by an intrinsic complexity that, from the very beginning, has required the usage of approximation techniques in their processing. In the first Geographic Information Systems (GISs) [46], approximation mainly concerned data capture and data

Alberto Belussi · Sara Migliorini

University of Verona, Italy

e-mail: alberto.belussi,sara.migliorini@univr.it

Barbara Catania

University of Genoa, Italy

e-mail: barbara.catania@unige.it

representation. As an example, the scale of a vector map, in cartography, implies a certain spatial accuracy of the spatial objects it contains. This accuracy regards absolute and relative positions of objects in the embedded space and has a strong influence in data processing since the result of a user query, while computed in a precise way, is always represented at a certain level of accuracy. In modern GISs, although the accuracy problem has not been completely solved, approximation has become a hot topic also for the processing itself, in order to improve query performance while returning a precise result [46]. Usual query processing techniques require the approximation of arbitrary geometric data with simpler objects, like rectangles or convex polygons [46]. Such simpler objects are then indexed and used to answer queries in two steps: in the first step, spatial data are filtered based on their simplified geometries, in order to detect which objects may with high probability satisfy the query (thus, an approximate result, containing the precise answer, is returned); in the second step, the exact geometry is taken into account in order to refine the result obtained in the first step and return to the user the precise answer to the query.

As discussed in Chapter 1, in the last years, there has been a rapid evolution of environments and applications that has radically modified query processing over data collections and has led to a re-interpretation of the concept of approximation, for traditional and spatial data. In such new environments, data is often heterogeneous and their characteristics highly variable. Heterogeneity in spatial data is often due to the employment of different resolution levels in representing data referring to the same geographical area or of different accuracy levels. While such data are often collected by different institutions and produced by different kinds of processes (e.g., social, ecological, economic), at different times, often they need to be handled together (as in Spatial Data Infrastructures) [45]; as a consequence, adequate integration approaches have therefore been provided [12, 24, 38]. Additionally, data heterogeneity and variability make query specification an issue since the user may not always be able to specify the query in a complete and exact way since she may not know all the characteristics of data to be queried, even if data come from just one single source (possibly, because such characteristics may change during query execution, as in mash-up applications [32]). Such problems are even more evident in distributed architectures, where input data may come from many different sources, in different formats, and can be made available by systems with different performance and availability. The resources for data integration and querying are in those cases often insufficient to get a precise response in a reasonable time and space.

In all the contexts cited above, it is too ambitious to get from query execution only precise answers, exactly satisfying the search condition expressed by the query, since in the above mentioned contexts it is quite common to find inconsistent or ambiguous data, it is really difficult to exactly characterize what we are looking for, and resources for a precise computation may not be available in a reasonable time. In particular, as a consequence of heterogeneity and limited knowledge about data, the quality of the obtained result, in terms of completeness and effectiveness, may be

reduced, since either interesting objects may not be returned (empty answer problem) or several uninteresting objects can be returned as answer, thus reducing user satisfaction (many answers problem) [2].

Based on the previous problems, the need thus arises of taking into account heterogeneity as well as limited user knowledge and resources in query specification and processing, with the aim of retrieving a relaxed or approximate solution while achieving a higher user satisfaction (see Chapters 2, 3, and 4). When considering such kind of techniques applied to spatial data, the focus is not on the approximate geometries to be returned as results, as occurred in the past monolithic systems, but on the (precise) result set which is either stretched, in order to include also objects that partially satisfy the query or are (spatially or in other sense) similar to the objects that exactly satisfy the request, or shrunk, in order to return only the objects which best-fit the request, based on some user preferences. Stretched results can also be returned in presence of very complex queries or limited resource availability, where it could be convenient to give up the target of producing the exact answer (since it would be too time-wasting) and focus on the retrieval of an approximate solution, using algorithms that often are order of magnitude less complex with respect to the ones producing a precise answer.

Techniques that approximate the query to be executed and not the data to be processed are called *query-based* in the following [1]. Due to their relevance for the development of a new generation of spatial applications, the aim of this chapter is to survey query-based approximation techniques for spatial data and to identify the issues that from our point of view have still to be investigated to complete the picture. To this aim, in Section 5.2 we first introduce background theory for spatial data, through a running example used in the remainder of the chapter for the description of the surveyed techniques. Section 5.3 classifies query-based approximation techniques for spatial data based on their aim, related data and query model, main action, information to be supplied for their application, and information used for approximating the result. Three main groups of techniques, namely top- k queries, skyline queries, and approximate query processing techniques, are then discussed for spatial data in Sections 5.4, 5.5, and 5.6, respectively. From the presented analysis, it follows that most of the proposed approaches for spatial approximate queries rely on the usage of quantitative, i.e., metric (distance-based), information. On the other hand, only few of them approximate queries taking into account qualitative information, for example by considering topological and cardinal spatial relations. Based on this consideration, in Section 5.7 we provide new types of queries based on qualitative relations, we discuss how the query processing algorithms already defined for metric relations can be extended to cope with qualitative relations as well as which topics require further research. Finally, Section 5.8 concludes the chapter by presenting some final remarks and discussing issues to be further investigated, based on what stated in Section 5.7.

¹ Query-based approximation approaches considered in this chapter include both QR and ApQP approaches as described in Chapter 1, excluding query personalization approaches, described in Chapter 4.

5.2 Background on Spatial Data and Queries

Based on the spatial relational models supported by the current GISs, a spatial database can be in general represented as a collection of tables (called *feature tables* or *maps* in the following). Each table contains tuples of values, also called *features* or *spatial objects*, and each value belongs to a specific domain. Besides atomic domains such as characters, strings, integer and real numbers, spatial domains described in the Simple Feature Model of OGC standards are typically available [31]. OGC primitive geometries include: (i) *Point*: a point is a 0-dimensional geometric object, representing a single location in the coordinate space; (ii) *Curve*: a curve is a 1-dimensional geometric object usually stored as a sequence of points, with the subtype of curve specifying the form of the interpolation between Points (for example, *LineString*, which uses linear interpolation between points); (iii) *Surface*: a surface is a 2-dimensional geometric object; a simple surface may consists of a single patch that is associated with one exterior boundary and 0 or more interior boundaries. A *Polygon* is a specialization of *Surface*, whose boundaries are *LineString* instances. In the following, if not otherwise stated, we assume each tuple has only one geometric attribute with domain: *Point*, *Curve* (or its specialization *LineString*), or *Surface* (or its specialization *Polygon*).

While some of the approaches we describe in the chapter have been defined for location-based applications, we will show they are relevant also for more typical GIS applications. To this aim, as running example, we consider a database containing the provinces, the municipalities, the rivers, the lakes, and the main towns of Northern Italy, with reference to the region around Venice (called Veneto). Each set of features is represented as a map. The spatial content of the database is shown in Figure 5.1. Data are modeled by introducing six different maps (tables), each having an attribute *geometry*, characterized by a spatial domain. Maps Mit_{Pr} , Mit_{Rv} , Mit_{Lk} , Mit_{Tw0} , Mit_{Tw2} , and Mit_{Mp} contain respectively: provinces, represented as polygons; main rivers, represented as lines; lakes, represented as polygons; main towns represented as points and as polygons; municipalities represented as polygons. Main rivers are labeled in the figure with their name; lakes are not labeled. Provinces are labeled by using their code and main towns by their name. Municipalities are not labeled. Feature tables, besides the geometric attribute, may contain other descriptive information. For example, each lake in Mit_{Lk} can be represented as a tuple with two attributes: one representing the lake name and the other its spatial extension.

Spatial data are usually related by spatial relations [46]. Topological, cardinal or distance-based relations cover almost all kinds of spatial relations usually available in real systems (even if OGC standards only support topological and distance-based relations). Topological relations are the most used and known relations in geographical applications. They are based on properties that are invariant with respect to homeomorphisms (i.e., transformations of the space, also called rubber sheet transformations, that stretch or shrink the space without cuttings or foldings). Many different definitions of topological relations have been proposed in literature. According to the one proposed by Egenhofer et al. [22], and taking into account the OGC standards, topological relations can be formally defined through the



Fig. 5.1 Datasets used in the running example: Provinces of Veneto (VR, VI, VE, BL, RO, TV) are filled in light pink; main rivers are drawn as dark blue lines and labeled with their name; lakes are in light blue; main towns are represented as points and also as polygons filled in green; municipality boundaries are green.

9-intersection model. In the 9-intersection model, each geometry A is represented by 3 point-sets: its interior A° , its exterior A^- , and its boundary ∂A . The definition of binary topological relations between two geometries A and B is based on the 9 intersections of each geometry component. Thus, a topological relation can be represented as a 3x3-matrix, called *9-intersection matrix*, defined as follows:

$$R(A, B) = \begin{pmatrix} A^\circ \cap B^\circ & A^\circ \cap \partial B & A^\circ \cap B^- \\ \partial A \cap B^\circ & \partial A \cap \partial B & \partial A \cap B^- \\ A^- \cap B^\circ & A^- \cap \partial B & A^- \cap B^- \end{pmatrix}$$

By considering the value empty (\emptyset) or not empty ($\neg\emptyset$) for each intersection, one can distinguish many relations between surfaces, lines, and points embedded in \mathbb{R}^2 . These relations are mutually exclusive and represent a complete coverage. In [15], this model has been extended by considering for each 9 intersection its dimension, giving raise to the *extended 9-intersection model*. Since the number of such relations is quite high, various partitions of the extended 9-intersection matrices have been proposed, grouping together similar matrices and assigning a name to each group. One of the widely used set of binary topological relations is the following: $\mathcal{T} = \{disjoint(d), touches(t), within(i), contains(c), equals(e), crosses(r),$

overlaps(o)} [15]. Such topological relations are those proposed by OGC and are available in many GISs products (e.g., PostGis)²

Cardinal (also called direction-based) and distance-based relations can be defined on top of topological relations. Indeed, a cardinal/distance-based relation between two geometries A and B can be defined by considering the topological relations that exist between A and a subdivision of the space induced by B [25, 53]. According to the type of the subdivision, a specific relation set is obtained. For cardinal relations, the subdivision induced by B produces the regions of the space that contain the points having the same cardinal relations with respect to B (e.g., North, Est, South and West or other more detailed cardinal sector sets). Such regions are called *tiles*. For example, when considering the North-South and Est-West directions, a region partitions the plane in 9 tiles, where the minimum bounding rectangle (MBR) of the region is the center tile. Based on such space subdivision, A is North with respect to B if A is *within* the sector that describes the region to the North of B .

For distance-based relations [46], the space is subdivided into two regions according to a given distance d : the set of points that have a distance from B less than or equal to d and the set of points that have a distance from B greater than d . For instance, A is within 3 km from B if A *overlaps* the circle representing the points that have a distance from B less than or equal to 3 km.

Spatial relations are the basis for the definition of spatial predicates, to be used in spatial query operators. Spatial selection, spatial join, nearest neighbor selection and join are the operators typically available in a spatial query language [46, 60]. More precisely, the following operators will be used in the paper examples:

- **Spatial selection (SS).** The spatial selection (σ) is applied to a map and returns the tuples of the map that satisfy a given selection condition. This condition includes spatial predicates (i.e., atomic formulas that include spatial relations, e.g., a topological relation, like *touches*). For example, we can select from map Mit_{Mp} the tuples representing municipalities that overlap a given rectangle R .
- **Spatial join (SJ).** The spatial join (\bowtie) is applied to a pair of maps M_1, M_2 and returns the tuples of the cross product between M_1, M_2 satisfying the spatial condition provided as join predicate. For example, we can join map Mit_{Mp} with map Mit_{Rv} , containing the municipalities and the main rivers of *Veneto*, respectively, in order to obtain a set of tuples each representing a municipality and the river it is crossed by.
- **Nearest neighbor selection (NN sel).** The nearest neighbor selection operation is applied to a map M and requires the specification of a spatial query object o . It returns the tuples in M having the minimum distance from o (usually the Euclidean distance is adopted, but other types of distance may also be used). For example, we can look for the municipalities that are closest to the point o representing a disaster event.
- **Nearest neighbor join (NN join).** The nearest neighbor join operation is applied to a pair of maps M_1, M_2 and returns the tuples of the cross product between $M_1,$

² <http://postgis.refractions.net/>

M_2 , each representing a feature f of M_1 and a feature g of M_2 , where g is one of the objects of M_2 nearest to f . For example, we can look for the lakes (contained in map Mit_{Lk}) and their closest municipalities (contained in map Mit_{Mp}).

5.3 A Taxonomy of Query-Based Approximation Techniques for Spatial Data

Query-based approximation techniques are traditionally classified into two main groups [13]: query relaxation (QR), including preference-based queries (PQ), and approximate query processing (ApQP). While QR and PQ have been mainly introduced with the aim of improving the quality of the query result, in presence of limited data knowledge during query specification and data heterogeneity, ApQP techniques have been proposed for either improving result quality (for example, by replacing equality checks with similarity checks in presence of highly heterogeneous data) or efficiency (for example, by avoiding the analysis of the overall datasets in order to improve performance).

In the following, QR and ApQP techniques are classified with respect to their main features and specific issues arising for spatial data are discussed in details. The result of the classification is presented in Table 5.1. The table provides a broad overview of each class of query approximation techniques proposed for spatial data we describe in more details in Sections 5.3.1 and 5.3.2. For each group of techniques, the following features are pointed out: (i) which type of data is considered (a specific geometric type or any geometric type); (ii) the spatial operators to which they can be applied (usually, spatial selection and join); (iii) the type of action (stretching the result, shrinking the result, extract the best); (iv) which information should be supplied by the user or the system in order to apply the corresponding technique; (v) which kind of information is considered for relaxation or approximation (spatial relationships, spatial/non-spatial attributes).

5.3.1 Query Relaxation

The concept of query relaxation (QR) has been introduced in Information Retrieval (IR) and adopted in several contexts as an approach for avoiding the empty or too many answers problems [2]. The main idea of QR is to modify the query in order to stretch or shrink the result set. QR approaches can be classified, depending on their scope, into query rewriting [6, 33, 35, 37] and preference-based approaches (e.g., top-k and skyline queries or preference-based query processing - see Chapters 2, 3, and 4 and references below).

Techniques based on *query rewriting approaches* rewrite the query using less or more strict operators, in order to get a larger or smaller answer set, and can be used to address both the empty answers and the many answers problems. Most query rewriting approaches for non spatial data rely either on information concerning data distribution and query size estimation (*value-based techniques*) or structure

Table 5.1 A summary of existing approximation techniques for spatial data.

		Data model		Query model		Action		Supplied Inform.		Type of used Information						
		Single geometric type	Any geometric type	Selection	Join	Shrink	Stretch	Get the best	Ranking function	Interesting attributes	Similarity function	Topological relations	Cardinal relations	Distance-based relations	Other spatial attributes	Non-spatial attributes
Query rewriting	NN and Thr top. relations [6]		✓	✓	✓		✓			✓	✓					
Top-<i>k</i> queries	Top- <i>k</i> spatial preference [59]	✓	✓	✓				✓	✓				✓			✓
	Top- <i>k</i> most influen. site [57]	✓	✓	✓				✓	✓				✓			✓
	Top- <i>k</i> spatial join [61]		✓		✓			✓	✓		✓					
Skyline queries	Spatial skyline, distance based [50] [51] [54]	✓		✓				✓		✓			✓			✓
	Spatial skyline, distance and direction based [26]	✓		✓				✓		✓		✓	✓			✓
Approx. query processing	Multiway spatial join [39]		✓		✓		✓				✓					
	Approximate distance-based queries [17] [20]	✓		✓	✓		✓						✓			
	Query Processing using Raster Signatures [3] [62]	✓		✓	✓		✓				✓		✓			

information (*structure-based techniques*). In value-based techniques, input information concerning the desired cardinality of the result set and information concerning data distribution are used to relax queries [33, 37]. On the other hand, most structure-based approaches have been proposed for semi-structured information like XML documents, where structure information may refer to the type of relationships existing between nodes and order information [35].

When considering query rewriting approaches for spatial data, some specific issues have to be considered: (i) spatial relations usually are not ordering relations; (ii) constants appearing in the queries are represented as values in multidimensional spaces. These considerations make value-based techniques and structure-based techniques not very suitable. Rather, relaxation based on properties of the used query conditions seem more adequate. In this case, a query predicate is rewritten into one or more different predicates in order to change the selectivity of the condition. All types of spatial predicates (topological, cardinal, distance-based) can be considered to this purpose. A proposal of query rewriting technique based on topological predicates is presented in [6].

In *preference-based approaches*, user or system preferences are taken into account in order to generate the result, with the aim of providing best results first. Such techniques can also be thought as a shrinking approach with respect to the overall set of possible results, since they reduce the cardinality of the ranked result dataset, or as a stretching approach with respect to the set of optimal results. In this category, we include both the top- k operators and the skyline operators.

The aim of a *top- k operator* is to restrict the number of returned results to a fixed number (k), based on some ranking function. Most top- k operators have been proposed for monotone ranking functions (see Chapter 3). This class of functions gives the opportunity of optimizing top- k query processing, using some threshold value to prune the visit of data. When considering spatial data, since spatial objects cannot be totally ordered while maintaining the properties of the space in which they are located, monotonicity is no more a relevant property. Rather, spatial relationships, and especially the distance-based ones, are often considered in computing scores. As a consequence, nearest neighbor operators are examples of top-1 queries, using distance as ranking function [16, 28, 48, 49].

Other approaches have then been proposed, using different types of ranking functions, based on spatial and non-spatial properties of spatial objects [57, 59, 61]. Such approaches differ for the reference data model, the used ranking function, and the provided processing algorithms. The most efficient algorithms assume that data are indexed and use a branch and bound approach to prune index subtrees that cannot provide any answer.

Skyline queries represent an alternative way to score objects based on a specific ranking function (whose definition may be cumbersome) by returning just the best results among all the possible ones (see Chapters 2 and 3). Best results can be defined in terms of a partial relation among objects and in terms of a dominance relation with respect to a set of given attributes (representing the user preference), by returning those objects that are not dominated by any other object (*skyline objects*).

Given a set of points, each corresponding to a list of values for the relevant attributes, a point A dominates a point B if it is better in at least one dimension and equal or better in all the others, with respect to some ordering [10]. More formally, assuming lower values are preferred, given two points $p(p_1, \dots, p_d)$ and $p'(p'_1, \dots, p'_d)$, p dominates p' if and only if we have $p_i \leq p'_i$ for $1 \leq i \leq d$ and $p_j < p'_j$ for some $1 \leq j \leq d$. Various algorithms have been proposed for skyline computation. Partition-based and index-based techniques (B-tree [10], bitmap [55], R-trees [34, 41]) avoid scanning the overall set of data for skyline computation, improving performance with respect to basic techniques.

When considering the spatial context, the concept of *spatial skyline* has been introduced. Given a set of data points P and a set of query points Q in a d -dimensional space, a spatial skyline query retrieves those points of P that are not dominated by any other point in P , with respect to a set of derived spatial attributes. Such attributes may refer to both spatial properties with respect to objects in Q (like distance [50, 51, 54] and direction [26]) and non-spatial attributes of P . Alternative

skyline definitions have also been provided in the context of road networks [56]. We do not further investigate these proposals in the sequel since more related to graph databases.

While top- k operators return a small result set at the price of specifying a ranking function, which is not a simple task, skyline operators avoid this specification at the price of a larger result set, which, even for two dimensional interest attributes, may be quite huge. In both cases, processing over join operations is challenging [30].

5.3.2 Approximate Query Processing

While QR deals with the specification of new (kind of) queries, approximate query processing (ApQP) refers to all the techniques for executing an intrinsically expensive traditional or relaxed query (e.g., a join or a top- k operator) by using ad hoc query processing algorithms, which automatically apply the minimum amount of relaxation based on the available data and resources, in order to efficiently compute a non-empty result similar to the user request. ApQP is suitable in all environments where data are huge, heterogeneous, and resources are limited or queries are complex [1, 4, 14]. Differently from QR approaches, in this case the query is not changed; rather, its execution is modified in order to get an approximate answer.

Traditional ApQP techniques for stored data essentially rely on four different approaches [3]. Under the first approach, existing processing algorithms are modified in order to generate an approximate result: the modification alters some steps of the algorithm by introducing weaker conditions (for instance, equality checks can be substituted by similarity checks in order to improve quality or, in a branch and bound approach, the pruning condition could be made more selective in order to prune the visit of a larger portion of the original dataset, thus improving time efficiency) (see, e.g., [52]). Under the second approach, new algorithms are provided, sometimes by reformulating the ApQP problem as a combination of non approximate operations (for instance, an approximate join on string attributes can be reduced to an aggregated set intersection, as shown in [58]). Under the third approach, heuristics are provided in order to reduce the computational cost of a given algorithm. They explore the solutions space in an effective way by exploiting some data properties (e.g., by looking for local optimal solutions) [43]. Genetic algorithms are included in this approach. All the first three categories of ApQP approaches do not alter the input dataset; on the other hand, the fourth group of techniques alters the input datasets each time it is too expensive to deal with values of a given domain or with the overall set of objects, by introducing approximate representations (e.g., synoptics or wavelets). Specific query processing algorithms are then provided to deal with these new datasets (see, e.g., [1, 4]).

As expected, also for spatial data ApQP techniques that represent the four approaches described above have been proposed (e.g., [16, 17, 20] for the first

³ The proposed classification extends that presented in Chapter 10 for approximate techniques having ‘processing algorithm’ as subject. Techniques here belonging to the fourth group are called *data reduction* techniques in Chapter 10.

approach, [39] for the third approach, and [3, 62] for the second and fourth approach). Such techniques have been provided for expensive operations like multi-way spatial join [18, 39], i.e., a join of more than two maps, and distance-based queries, including several variations of the NN queries presented in Section 5.2 which become quite inefficient in high-dimensional spaces [39]. Since spatial data processing relies on the usage of index structures (e.g., R-trees [27]), such structures have also been extended to cope with the approximate processing. Concerning the fourth approach, due to the high computational complexity in manipulating spatial values, techniques are usually provided introducing an approximate representation of the exact geometries of objects, potentially completely different with respect to the one used for the exact geometries (for example, the exact geometry is in vector format, while the approximation is a grid of cells or a string of bits) [3, 62]. In this case, new algorithms have to be designed in order to produce an approximate result from the processing of traditional operations, by manipulating only the approximate representations. Differently from solutions based on traditional approximate data representation, as MBRs, such representations have been provided specifically for approximate processing and in general they do not allow one to detect the precise query result.

We remark that many other papers concerning spatial data include the word approximation in their claims but are specifically dedicated to a particular application domain where time is relevant. For instance, many efforts have been devoted to the problem of finding the approximate result of a k NN selection considering a moving object as query point [29]. Here the approximation is due to the fact the the query point is moving and we do not have the exact knowledge about its position that changes over time. Another interesting issue, we are not considering in this paper, deals with the similarity of trajectories of moving objects [5], which indeed use some sort of approximation in the definition of similarity among trajectories. Other papers just integrate other kinds of approximate queries with a spatial predicate; for instance, in [58] the authors propose a data access structure that integrates the execution of an approximate string search with spatial queries, like k NN and range-based selection.

5.4 Spatial Top- k Queries

A top- k operator returns the first k objects according to some ranking function and some ordering among ranking values. As already pointed out in Section 5.3, various top- k operators have been proposed for spatial data and various query processing algorithms have been developed for them. Besides the general features already considered in Section 5.3, top- k operators can be classified depending on: (i) the used ranking function; (ii) the used query processing algorithm. In the following, existing proposals will be classified based on these two parameters.

5.4.1 Top- k Ranking Function

In the context of spatial data, three main kinds of ranking functions have been proposed (see Table 5.1). They use in different ways object distances, intersections, and spatial and non-spatial object properties to compute scores.

Top- k spatial preference queries. A top- k spatial preference query selects the k best spatial objects (which are points in the proposal but can be extended to any type of objects) with respect to the quality of features in their spatial neighborhood [59]. Assume that objects in the space are qualified by several features, leading to m feature sets F_1, \dots, F_m . The *score* of an object thus quantifies the quality ω of features, which may belong to distinct datasets, in its spatial neighborhood (defined either through a distance-based selection or a nearest neighbor operator). For each feature set, the score of the selected features is computed and all the scores are then aggregated to generate the overall score for the given object. Quality can be assessed by considering either spatial or non-spatial properties of features. The following definitions refer to points but they can be easily extended to cope with objects of an arbitrary geometric type.

Definition 1 (Top- k spatial preference ranking function [59]). Let S be a point dataset and F_1, \dots, F_m datasets of features. The score of a point $p \in S$ is defined as follows:

$$\tau^\theta(p) = \text{agg}\{\tau_i^\theta(p) \mid i \in [1, m]\} \quad (5.1)$$

where *agg* is a monotone aggregate operator and $\tau_i^\theta(p)$ is the i -th component score of p with respect to the neighborhood condition θ and the i -th feature dataset F_i . \square

Typical examples of the aggregate function *agg* are: SUM, MIN and MAX. The neighborhood condition θ defines the spatial neighborhood region of a point p that has to be considered during the computation of the score function. Two conditions have been considered in [59]: (i) range condition *rng*, which considers all features that are within a parameter distance ε from p ; (ii) nearest neighbor condition *nn*, which considers nearest neighbor features of p . Such conditions give rise to different scores τ_i^θ :

- the *range score* $\tau_i^{\text{rng}}(p)$ returns the maximum quality $\omega(s)$ of features $s \in F_i$ which satisfy the range condition with respect to p (i.e., that are within a parameter distance ε_i from p) or 0 if no such point exists;
- the *nearest neighbor score* $\tau_i^{\text{nn}}(p)$ returns the quality $\omega(s)$ of feature $s \in F_i$, which satisfies the nearest neighbor condition with respect to p (i.e., s is the nearest neighbor of p in F_i).

Example 1. Consider the scenario presented in Section 5.2. The following is an example of top- k spatial preference query: “Retrieve the 2 main towns (as polygons) in Veneto with the highest score, computed as the maximum area of the lakes that lie within 25 km”. Based on Definition 1, the aggregate function used in modeling

the previous request is MAX. A single feature set F_1 is considered, corresponding to lakes, the considered quality function ω is the area, a range score is computed with respect to objects within 25 km from each municipality. \diamond

Top- k influential sites. Another ranking function has been proposed by considering an extension of the Reverse Nearest Neighbor (RNN) problem, called *bichromatic RNN* [57]. Given two sets of points, S (sites) and O (input objects), the bichromatic RNN of a site s (also called the *influence set* of s) is the set of points in O that have s as the nearest site. If each spatial object in O has a weight ω , the score of a site is computed as the sum of the weight of objects in its influence set. While the ranking function has been originally defined for points, it can be extended to cope with arbitrary geometric types.

Definition 2 (Top- k influence site ranking function [57]). Let S and O be two sets of points, representing sites and input spatial objects, respectively. The score of a site $p \in S$ is defined as follows:

$$\tau(p) = \text{sum}\{\omega(o) \mid o \text{ belongs to the influence set of } p\} \quad \square$$

Example 2. Consider the scenario presented in Section 5.2 and suppose that: the query is defined on the Veneto region, the set of spatial objects O is composed of the municipalities in this region, the set of sites S contains the main hospitals as points (located in the main towns: *Verona, Vicenza, Padova, Venezia, Belluno, Treviso* and *Rovigo*), and the weight associated with each object in O is the municipality inhabitants. A top- k most influential site query may return the first k most influential hospitals, namely the hospitals for which the total number of inhabitants, with respect to the municipalities that are closer to such hospitals than to any other, is the highest. \diamond

Top- k spatial join. In the context of the top- k spatial join, a ranking function based on the number of intersections between pairs of spatial objects has been provided [61]. More precisely, given two datasets A and B , for each spatial object in A or B , the ranking function returns the number of intersections with objects in the other dataset.

Definition 3 (Top- k Spatial Join ranking function [61]). Let A and B two spatial datasets. Let $o \in C$, let $C \in \{A, B\}$ (i.e., C is the set to which o belongs among A and B). The score of o is computed as follows:

$$\tau(o) = \text{card}\{o' \mid o' \in (A \cup B - C), o \text{ intersects } o'\}. \quad \square$$

Example 3. A top- k spatial join over the river and the municipality datasets (maps Mit_{R_v} and Mit_{M_p}) may return the rivers crossing the maximum number of municipalities or the municipalities crossed by the maximum number of rivers. With the datasets of the running example, as shown in Figure 5.1 with high probability the

k answers will correspond to k rivers, since the municipalities crossing a river are many more than the rivers that cross a municipality. \diamond

5.4.2 Top- k Query Processing Algorithms

As already pointed out, monotonicity of ranking functions for traditional data is a relevant assumption since it allows the optimization of top- k query processing through the usage of specific threshold values. Data can be either sequentially accessed, in an ordered way with respect to each quality used in score computation, or randomly accessed, through the usage of some index structure. The monotonicity assumption allows the query processor to maintain some upper/lower bounds on the score values of objects not yet visited. Using such bounds, the computation can be stopped as soon as it is clear that no more results can be found.

A similar approach does not work for spatial data. While for traditional data an ordered sequential access is an option, for spatial data an index-based access (based on R-trees or other spatial data structures) is usually a must.⁴ Additionally, as discussed in the previous section, ranking functions for spatial data rely on spatial relationships. New approaches have therefore been designed for computing top- k spatial queries in an optimized way.

Even if spatial objects inside a spatial index are not totally ordered, they are however spatially organized and properties of such organization can be used to optimize top- k processing. Whatever visit is used, similarly to the traditional context, the following structures are usually maintained during top- k result computation:⁵ (i) a queue Q , used to store top- k results generated so far, ordered with respect to their score value; (ii) a threshold value ρ , representing the lowest score value in Q (i.e., a lower bound of the k highest scores based on objects visited so far); (iii) specific data structures to efficiently compute the object scores.

Algorithm 1. Algorithm for top- k traditional visit (TV).

INPUT: The node N of the spatial data index structure to be processed.

The queue Q used to store the top- k results generated so far, it is initially empty.

The threshold value ρ representing the lowest score value in Q , it is initially equal to 0.

OUTPUT: An updated queue Q containing the top- k results.

```

1. Algorithm  $TV(N, Q, \rho)$ 
2. for all entries  $e$  in  $N$  do
3.   if  $N$  is non-leaf then
4.     read the child node  $N'$  pointed by  $e$ ;
5.      $TV(N', Q, \rho)$ ;
6.   else
7.     compute  $score(e)$ 
8.     if  $score(e) \geq \rho$  then
9.       update  $Q$  and  $\rho$ 
10.    end if
11.  end if
12. end for

```

⁴ For simplicity, in the following we assume that leaf nodes of the index tree contain objects and not their approximations.

⁵ In the following, we assume that highest values are preferred.

Algorithm 2. Algorithm for top- k Branch and Bound visit: local ordering (BB1).**INPUT:** The node N of the spatial data index structure to be processed.The queue Q used to store the top- k results generated so far, it is initially empty.The threshold value ρ representing the lowest score value in Q , it is initially equal to 0.**OUTPUT:** An updated queue Q containing the top- k results.

```

1. Algorithm  $BB1(N, Q, \rho)$ 
2.  $V := \{e | e \text{ is an entry in } N\}$ ;
3. if  $N$  is non-leaf then
4.   for all entries  $e$  in  $N$  do
5.     compute  $key(e)$ ;
6.     remove  $e$  from  $V$  if  $key(e) < \rho$ ;
7.   end for
8.   order  $V$  in descending order, according to key values just computed
9.   for all entries  $e$  in  $V$  (accessed in an ordered way) do
10.    read node  $N'$  pointed by  $e$ 
11.     $BB1(N', Q, \rho)$ 
12.   end for
13. else
14.   for all entries  $e$  in  $N$  do
15.     compute  $score(e)$ ;
16.     remove  $e$  from  $V$  if  $score(e) < \rho$ ;
17.   end for
18.   update  $Q$  and  $\rho$  according to entries in  $V$ ;
19. end if

```

Two main types of visits can be implemented:

- *Traditional visit.* Under this approach, the spatial index is visited usually in depth-first or breadth-first way [59]. As an example, Algorithm 1 for top- k traditional visit relies on a depth first search. During the visit, the *score* is computed for objects pointed by leaf entries. Thus, Q will contain objects that, based on the entries visited so far, belong to the top- k result. If, during the visit, the score of an object o is higher than or equal to ρ , then ρ and Q are updated. More precisely, o is inserted in Q and all objects that do not belong any more to the top- k result are removed from Q .
- *Branch and Bound visit (BB).* Under the Branch and Bound visit, the visit of a subtree is avoided if and only if it is possible to establish that objects pointed by its leaves do not belong to the result [28, 59, 61]. To guarantee efficiency, such property has to be checked locally at the index entry under consideration, using some *key value*. The key value usually corresponds to an upper bound of the score of the objects belonging to the subtree rooted by the node pointed by the considered index entry. The visit of the subtree can be avoided if the key value is lower than ρ . An *ordering value* is also used for determining the ordering upon which entries are visited, with the aim of visiting first the entries that most probably will generate some results. In many cases, the value upon which the ordering is defined coincides with the key value. The ordering can be applied either to all the entries in each visited node (*local ordering*) or to all the entries/objects visited so far (*global ordering*). In this second case, Q contains not only objects that, based on the entries visited so far, belong to the top- k result, but also entries (or nodes) that may potentially produce further results. Additionally, the key value for an entry corresponds to an interval containing all the key values associated with objects in the subtree rooted by the considered entry. As an example,

Algorithm 2 denoted as BB1, relies on the local ordering, while Algorithm 3 called BB2, relies on the global ordering. Under BB2, when an entry is considered, if based on the key value it is possible to determine that further results can be produced by such entry, the entry is expanded, i.e., it is replaced by the entries of the pointed node. While BB2 provides a progressive generation of results, in BB1 (and in TV) results correspond to the objects contained in Q at the end of the processing.

We notice that the ability to prune index subtrees in BB comes at the price of managing a potentially longer Q with respect to TV , since in BB Q may contain both objects and entries.

Query processing algorithms provided for computing top- k results based on the ranking functions proposed in Section 5.4.1 instantiate the general algorithms presented above. Table 5.2 summarizes the characteristics of such algorithms, based on the general description provided above. In the following, they will be shortly described, with respect to the type of visit they implement (either traditional or branch and bound).

Algorithm 3. Algorithm for top- k Branch and Bound visit: global ordering (BB2).

INPUT: The spatial data index structure R .
OUTPUT: The queue Q containing the top- k results.

1. **Algorithm** $BB2(R)$
2. PriorityQueue $Q := \emptyset$
3. $Q.insert(R.root, 0)$, where 0 is the priority associated to $R.root$
4. $\rho = 0$
5. **while** Q is not empty and the number of reported elements is lower than k **do**
6. $N := Q.getMax()$
7. **if** N is non-leaf **then**
8. **for** all entries e in N **do**
9. compute $key(e)$;
10. **if** $key(e).max \geq \rho$ **then**
11. $Q.insert(N', key(e))$, where N' is the node pointed by e ;
12. update Q and ρ according to e ;
13. **end if**
14. **end for**
15. **else if** N is leaf **then**
16. **for** all entries e in N **do**
17. compute $score(e)$;
18. **if** $score(e).max \geq \rho$ **then**
19. $Q.insert(o, score(e))$, where o is the object pointed by e ;
20. update Q and ρ according to e ;
21. **end if**
22. **end for**
23. **else**
24. return N ;
25. **end if**
26. **end while**

5.4.2.1 Traditional Visit

Top- k Spatial Preference Query. The traditional visit proposed in [59] for top- k preference queries with MAX as aggregate function and descending score ordering instantiates Algorithm 1. In order to compute the score of an object, the technique assumes that each feature dataset F_i is indexed by a max aggregate R-Tree aR_i [40].

Table 5.2 A summary of existing top- k spatial query processing approaches.

		Specific data structures for score computation	Key value	Ordering value
Traditional visit	Top- k preference query: probing algorithm [59]	aR-Tree aR_i for each feature dataset F_i		-
Branch and Bound visit	Top- k preference query: Branch and Bound algorithm [59]	aR-Tree aR_i for each feature dataset F_i	$T(e)$: score upper bound for the objects in the subtree rooted by e relying on the computation of the Minkowski region	
	Top- k preference query: Feature Join algorithm [59]	heap H of feature combinations fc_i aR-Tree aR_i for each feature dataset F_i	$T(e)$ for R , as above $\tau(fc_i)$ for H : aggregation of the quality of each feature in fc_i	
	Top- k most influential sites [57]	$queue_{s_{in}}$ entries of R_i inside query region $queue_{s_{out}}$ entries of R_i outside query region $queue_o$ entries of R_o that effects $queue_{s_{in}}$	$maxInfluence$ for $queue_{s_{in}}$ $minExistDNN$ for $queue_{s_{out}}$	$maxInfluence$ for $queue_{s_{in}}$
	Top- k spatial join [61]	intersection list $IL(e)$ for each entry e of R-trees R_a and R_b , indexing the two input datasets	$count(e)$: number of the entries intersecting e and belonging to the other dataset	

This is an R-tree where each non-leaf entry is augmented with the maximum feature quality value contained in its subtree. For computing the score of a leaf entry in the R-tree indexing the input dataset, for each feature dataset F_i , the aR-tree aR_i is accessed and the score is computed by executing either a range search or a NN search, depending on the type of the query (range score or nearest neighbor score) and on the leaf entry at hand. Of course, this approach is inefficient for large datasets, because the score of different objects is separately computed, thus the aR-Tree of the feature datasets has to be traversed many times. A more efficient variant has therefore been proposed, called *group probing algorithm*, in which the score of the objects in the same leaf node of the R-Tree are computed concurrently with a single traversal of each aR_i tree.

Example 4. Consider the scenario presented in Section 5.2 and the query proposed in Example 1: “Retrieve the 2 main towns (as polygons) in Veneto with the highest score, computed as the maximum area of the lakes that lie within 25 km”. The feature dataset, corresponding to lakes, is indexed by an aR-tree S , where each non-leaf entry of S is augmented with the maximum area of the lakes in its subtree, while each leaf entry is annotated with the area of the corresponding lake. The query requires the computation of a range score, with $\varepsilon = 25$ km. Figure 5.2 shows the R-Tree R of the input dataset and the aR-Tree S for lakes. Using Algorithm 1 the nodes of R are visited in depth-first order: $[R_1, R_2, R_7, R_8, R_3, R_5, R_6, R_4, R_9, R_{10}, R_{11}]$ and the score of each leaf is computed using the aR-Tree S by executing a range query. The query object of the range query corresponds to the buffer of the leaf object, computed with distance 25 km. Initially, the queue Q is empty and the threshold ρ is equal to zero, then for each leaf in R , Q and ρ are updated as in Table 5.3. Notice that the threshold ρ remains equal to zero until k (in this case two) objects are loaded onto the queue. The result corresponds to entries R_7 and R_5 , i.e., Verona and Belluno. \diamond

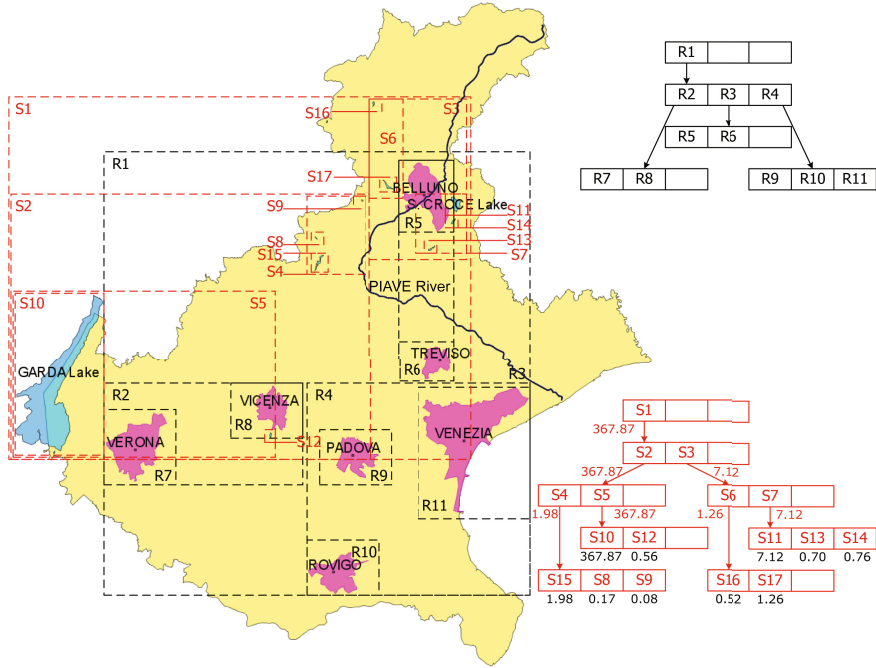


Fig. 5.2 R-Tree R of the input dataset (i.e., main towns in Veneto) and aR-Tree S of lakes, considered in Example 4

Table 5.3 Example of execution of Algorithm TV for a top- k spatial preference query. Columns “ Q BL9”, “ Q AL9”, “ ρ BL9” and “ ρ AL9” contain the value of Q and ρ before and after executing line 9 of Algorithm 11 respectively, i.e., before and after updating Q and ρ for each visited entry.

Entry e	Main Town	$score(e)$	Q BL9	ρ BL9	Q AL9	ρ AL9
R_7	Verona	367.87	\square	0	$[R_7]$	367.87
R_8	Vicenza	0.56	$[R_7, null]$	0	$[R_7, R_8]$	0.56
R_5	Belluno	7.12	$[R_7, R_8]$	0.56	$[R_7, R_5]$	7.12
R_6	Treviso	0	$[R_7, R_5]$	7.12	$[R_7, R_5]$	7.12
R_9	Padova	0.56	$[R_7, R_5]$	7.12	$[R_7, R_5]$	7.12
R_{10}	Rovigo	0	$[R_7, R_5]$	7.12	$[R_7, R_5]$	7.12
R_{11}	Venezia	0	$[R_7, R_5]$	7.12	$[R_7, R_5]$	7.12

5.4.2.2 Branch and Bound Visit

Top- k Spatial Preference Query. The branch and bound algorithm proposed in [59] for top- k preference queries relies on the algorithm scheme BB1. In order to compute the score, each feature dataset F_i is indexed as described in Section 5.4.2.1. The key value of an entry e of the R-Tree R indexing the input dataset coincides with an upper bound $T(e)$ of the score of objects contained in the subtree rooted by e . $T(e)$

is also used for ordering purposes. The computation of such bounds depends on the type of the considered query (either range or nearest neighbor queries) and relies on the computation of the Minkowsky region [9]⁶ of leaf entries in the R-tree. More precisely, the key value $T(e)$ of each non leaf entry e in R is the maximum quality of the lowest level of non-leaf entries of S intersecting the Minkowski region of the MBR associated with the current entry e of R .⁷ Score computation coincides with that described for the traditional visit and suffers of the problems pointed out in Section 5.4.2.1. An alternative method, which can be still classified as branch and bound and is called *Feature join*, has therefore been presented. It consists in performing a multiway spatial join of the feature datasets F_1, \dots, F_m to obtain combinations of features that with a high probability will be in the neighborhood of some object in the dataset D . Combinations are lists of nodes from aR_1, \dots, aR_m , respectively, and are maintained in a max-heap H in order of their score. The score of a combination is denoted by $\tau(fc_i)$ and is computed as the aggregation of the quality of each feature in the combination. At each iteration, the spatial region in H with the highest score is examined, in order to find data objects in D having the currently considered feature combination in their neighborhood. The proposed algorithm applies a branch and bound visit to both feature trees and object tree. In the first case, the key and ordering value for entries in H are represented by the score of each feature combination. In the second case, key and ordering value correspond to $T(e)$, slightly modified to take into account clear cases of non interesting visits.

It has been shown that, in order to execute top-k preference queries, the branch and bound visit is the best method when the object dataset is small whereas the Feature join algorithm is the best algorithm when there are few and small feature datasets.

Top-k Most Influential Sites. The algorithm proposed in [57] computes the top-k most influential sites which are contained in a query region P . Such algorithm does not rely on the schemes TV, BB1, and BB2 presented above, due to the nature of the problems (two datasets, sites and objects, are considered and indexed through two R-trees R_s and R_o). However, it can still be considered a branch and bound approach since the visit of such trees is pruned based on some heuristics in order to avoid to search useless portions of the trees. To this aim, three queues are maintained, which contain index and leaf entries (thus, in some sense, the algorithm adheres to the BB2 scheme):

- $queue_{s_{in}}$, which keeps the (intermediate or leaf) entries of R_s that are inside the query region P ; each entry S_j is associated with an interval (the key value for S_j) ($minInfluence, maxInfluence$) corresponding to the influence range of sites indexed below S_j . Such values are computed based on the weight of entries in $queue_o$ (see below) that affect the considered entry in $queue_{s_{in}}$. Potential answers come from entries in such queue.

⁶ The Minkowski region with respect to ϵ of an object o is the set of points whose minimum distance from o is within a distance ϵ .

⁷ The choice of the lowest level non-leaf entries seems to guarantee the best trade-off between computational cost and result quality, according to [59].

- $queue_o$, which keeps the (intermediate or leaf) entries of R_o that affects some entry in $queue_{sin}$. An entry e_o of R_o affects an entry e_s of R_s if some objects in the subtree rooted by e_o considers some sites in e_s as the closest site among the sites indexed by entries already contained in $queue_{sin}$ and $queue_{sou}$. The aim of entries in $queue_o$ is to compute ($minInfluence, maxInfluence$) intervals for entries in $queue_{sin}$.
- $queue_{sou}$, which keeps the (intermediate or leaf) entries of R_s that are outside the query region P but are affected by some entry in $queue_o$.

The algorithm starts by initializing the three queues using the root nodes of R_s and R_o . The elements in $queue_{sin}$ are progressively expanded (i.e., the entry is replaced by the entries contained in the pointed node) in order of their $maxInfluence$ value, until their actual influence is computed (i.e., $maxInfluence$ and $minInfluence$ become equal). In this way, the entries whose corresponding subtree may contain some of the k most influential site are visited first. An entry S_j with $maxInfluence = 0$ can be removed, since all sites indexed by S_j will have $influence = 0$, thus they cannot belong to the result. If a queue entry e_s in $queue_{sin}$ is not affected by any entry e_o in $queue_o$, it is removed as well. On the other hand, entries in $queue_o$ are progressively expanded only if they affect some entries in the other two queues.

In order to determine which entry in $queue_o$ affects the entries in $queue_{sin}$ and $queue_{sou}$, some pruning conditions have been proposed based on a new metric called $minExistDNN_{e_s}(e_o)$, which represents the smallest distance that is an upper bound of the distance between any object in e_o and its nearest site in e_s . Within such distance, any object in e_o can reach a site in e_s . Given an object entry e_o and two site entries e_s and e'_s , if $minExistDNN_{e_s}(e_o)$ is lower than the lower bound of the distance between any object in e_o to any site in e'_s , then e_o does not affect e'_s .

The algorithm terminates when there are k sites in $queue_{sin}$ whose $minInfluence$ is no less than $maxInfluence$ of all the remaining sites. We notice that the algorithm can return a site without computing its actual influence. Indeed, as long as the $minInfluence$ of a site is big enough, the site can be returned as output.

Top- k Spatial Join: TS Algorithm. A branch and bound algorithm has also been proposed for the top- k spatial join in [61]. The proposed algorithm is a variation of a classical join algorithm based on R-trees and adapts the algorithm scheme BB2 to the join case. The two input datasets A and B are indexed by two R-Trees R_a and R_b , respectively, which for simplicity are assumed to have the same height. The proposed algorithm inserts in the queue Q all objects or intermediate entries visited so far, which may contribute to the results.

The key value $key(e)$ associated with each entry e corresponds to the number of entries of the other tree intersecting e (also denoted by $count(e)$), together with the list of such entries (*intersection list* ($IL(e)$)). If e is a leaf entry (i.e., an object) of the R-Tree R_a , $count(e)$ is the number of objects of R_b that intersect e , thus it corresponds to $score(e)$. If e is an intermediate entry, $count(e)$ is an upper

bound of the actual count of any object in e , which is obtained as $count(e) = \sum_{e_i \in R_b \wedge e_i \text{ intersect } e} maxNum(e_i)$, where $maxNum(e_i)$ is estimated as $C^{level(e)}$, where C is the node capacity and $level(e)$ is the level of the node that contains e .

The algorithm starts from the root of the two trees R_a and R_b and computes all the pairs (e_a, e_b) of intersecting entries, each contained in one of the roots. For each entry e that appears in a pair, it then computes $key(e)$. The tuple $\langle e, count(e), IL(e) \rangle$ is inserted into a heap Q sorted by the counts in descending order (thus, $count$ is used both for keys and ordering). We do not need to visit entries whose $count$ is smaller than that of the top- k results already found (stored in ρ). We notice that, while in this case Q contains information about entries and not about nodes, the algorithm adheres to the general scheme BB2, presented in Subsection 5.4.2

Until the number of reported objects is less than k , the algorithm dequeues the first entry e of Q . If e is a leaf entry (i.e., an object) it is reported as result. If it is an intermediate entry pointing to a node n , for each node n_i pointed by an entry in $IL(e)$, the join between n and n_i is computed. For each entry e' in n , based on the detected intersecting pairs, $count(e')$ is computed and $IL(e')$ updated. If $count(e')$ is greater than the count of the k -th best object found so far, the tuple $\langle e', count(e'), IL(e') \rangle$ is inserted in Q . If it is a leaf entry, ρ is also updated.

Example 5. In order to show an example of usage of branch and bound algorithms, let us consider data represented in Figure 5.2, the top- k spatial preference query discussed in Example 4 and the algorithm scheme BB1, as suggested in [59].

The result of the BB1 algorithm execution is presented in Table 5.4. The column “ S entries” contains the entries of the aR-tree S (corresponding to the lake features) involved in the computation of $key(e)$, i.e., the entries which intersect the Minkowski region of the MBR associated with e , with respect to 25 km. $key(e)$ coincides with the highest quality of the S entries involved in the computation, i.e., the maximum area of the lakes which lie within 25 km from e . Entries e in V are always visited in the order of their key value. Notice that set V is local at each call of the BB1 algorithm and that the algorithm is recursively called for each entry of non-leaf nodes.

The result of the computation, according to Algorithm scheme BB2, is presented in Table 5.5. In this case, entry keys are represented by intervals, corresponding to a lower and an upper bound of score values contained in the subtree rooted by the considered entry. The upper bound of an entry e is computed as for BB1 while the lower bound can be computed as the minimum of the maximum areas associated with each S entry involved in the computation, as described above. The priority queue Q is a global variable containing either nodes or objects that may potentially give some results. Notice that, differently from BB1, entry R_4 is pruned.

A similar approach can be used to compute top- k spatial join. In this case, the queue will contain pairs of either entries or objects and a different key value is used.

On the other hand, the computation of top- k most influential sites relies on a different (but branch and bound) approach. We refer the reader to [57] for examples of its applications. \diamond

Table 5.4 Example of execution of Algorithm BB1 for a top- k spatial preference query.

N	e	S entries	$key(e)$	$score(e)$	V	Q	ρ
R_1					$\{R_2, R_3, R_4\}$	\square	0
	R_2	$\{S_5\}$	367.87	–	–	–	–
	R_3	$\{S_4, S_6, S_7\}$	7.12	–	–	–	–
	R_4	\emptyset	0	–	–	–	–
					$\{R_2, R_3, R_4\}$	\square	0
R_2					$\{R_7, R_8\}$	\square	0
	R_7	–	–	367.87	–	–	–
	R_8	–	–	0.56	–	–	–
					$\{R_7, R_8\}$	$[R_7, R_8]$	0.56
R_3					$\{R_5, R_6\}$	$[R_7, R_8]$	0.56
	R_5	–	–	7.12	–	–	–
	R_6	–	–	0	–	–	–
					$\{R_5\}$	$[R_7, R_5]$	7.12
R_4					$\{R_9, R_{10}, R_{11}\}$	$[R_7, R_5]$	7.12
	R_9	–	–	0	–	–	–
	R_{10}	–	–	0	–	–	–
	R_{11}	–	–	0	–	–	–
					\emptyset	$[R_7, R_5]$	7.12

Table 5.5 Example of execution of Algorithm BB2 for a top- k spatial preference query.

N	e	S entries	$key(e)$	$score(e)$	Q	ρ	$out\ put$
R_1					$[R_1]$	0	
					\square	0	
	R_2	$\{S_5\}$	$[367.87, 367.87]$	–	$[R_2]$	367.87	–
	R_3	$\{S_4, S_6, S_7\}$	$[1.26, 7.12]$	–	$[R_2, R_3]$	1.26	–
R_4	\emptyset	$[0, 0]$	–	$[R_2, R_3]$	1.26	–	
R_2					$[R_3]$	1.26	
	R_7	–	–	367.87	$[R_7, R_3]$	1.26	–
	R_8	–	–	0.56	$[R_7, R_3]$	1.26	–
R_7					$[R_3]$	1.26	$R_7 (367.87)$
R_3					\square	1.26	
	R_5	–	–	7.12	$[R_5]$	7.12	–
	R_6	–	–	0	$[R_5]$	7.12	–
R_5					\square	7.12	$R_5 (7.12)$

5.5 Spatial Skyline Queries

When considering spatial data, the notion of skyline query has been redefined by considering dimensions in the space as attributes of the objects. In the following, the concept of spatial skyline queries is introduced and related query processing issues discussed.

5.5.1 Spatial Skyline Queries

The concept of Spatial Skyline Query has been first proposed in [51], by extending the classical skyline query to the case where attribute values quantify spatial relationships between each input object and a given set of query objects, using a distance metric. In order to formally introduce spatial skyline queries, we first revise the concept of dominance in the spatial context.

Definition 4 (Spatial Dominance [51]). Let P be a set of spatial objects, Q a set of spatial query objects. Let $D(.,.)$ be a distance metric, satisfying the triangular inequality. A spatial object $p \in P$ spatially dominates another object $p' \in P$ with respect to Q if and only if:

- $D(p, q_i) \leq D(p', q_i)$ for all $q_i \in Q$, and
- $D(p, q_j) < D(p', q_j)$ for some $q_j \in Q$. □

The distance metric can be any function that obeys the triangular inequality, in order to guarantee the transitive property of the dominance. The Euclidean distance between two points and the Hausdorff distance [47] between two polygons or curves are examples of distance metric satisfying the triangular inequality. Given such definition of spatial dominance, the concept of skyline query can be easily extended to the spatial context as follows.

Definition 5 (Spatial Skyline Query [51]). Let P be a set of spatial objects, Q a set of spatial query objects. Let $D(.,.)$ be a distance metric. A spatial skyline query returns the subset $S \subseteq P$, called *spatial skyline*, of objects that are not spatially dominated by any other object in P with respect to Q . □

Example 6. Consider the spatial database introduced in Section 5.2 and a spatial query asking for all the main towns (represented as polygons) in Veneto that are at distance zero from both the rivers *Adige* and *Brenta*. None of the main towns in Figure 5.1 touches both the desired rivers, so an exact query will return an empty result set. On the contrary, a spatial skyline query with respect to the set of query objects $\{Adige, Brenta\}$ returns the set of main towns that are closest to both rivers. Thus, the set $\{Verona, Padova\}$ will be returned. These main towns are not dominated by any other main town, because no other main town is closer to the river *Adige* than *Verona*, and no other main town is closer to the river *Brenta* than *Padova*. ◇

The previous example introduces an important property of (spatial) skyline.

Property 1. [51] Given a set of data objects P and a set of query objects Q , if $p \in P$ is the unique closest point of P to $q \in Q$, then p belongs to the skyline of Q . ◇

It is important to remark that the concept of spatial skyline can be seen as a special case of the *dynamic skyline query*, introduced in [41]. A dynamic skyline query first maps each original point p into a new point p' such that $p' = (f_1(p), \dots, f_d(p))$, where f_i is a function defined upon p coordinates, and then computes the skyline of

this new set of data. The spatial skyline can be seen as a special case of dynamic skyline where $f_i = D(p, q_i)$, $1 \leq i \leq d$.

The notion of spatial skyline relies on a single distance metric. An extension of this concept has been proposed in [26] by considering, besides distance with respect to a single point q , also direction in detecting the best objects to be returned. It is therefore called *direction-based spatial skyline*. Such definition is quite interesting in location-based services (mobile recommendations, car navigations, etc.), where q corresponds to the user position, but, as we will show below, it may be useful also in more traditional spatial domains. The direction-based spatial skyline query returns all nearest objects around the user from different directions. Each object p in the skyline is nearer to q than any other objects having the same direction of p . The notion of direction has been formalized by considering the angle between the unit vector $(0,1)$ and the vector between the point p and the query object q . Actually, directions are not compared by equality but by similarity, by considering an input tolerance value θ .

Definition 6 (Direction-based Spatial Dominance [26]). Let P be a set of points, q a query point, and $D(.,.)$ a distance metric. A spatial object $p \in P$ D -spatially dominates another objects $p' \in P$ if and only if:

- p and p' are located in the same direction with respect to q , and
- $D(p, q) < D(p', q)$. □

Definition 7 (Direction-based Spatial Skyline Query [26]). Let P be a set of points, q a query point, and $D(.,.)$ a distance metric. A direction-based spatial skyline query returns the subset $S \subseteq P$, called *direction-based spatial skyline*, of objects that are not D -spatially dominated by any other object in P with respect to q . □

Example 7. Let us consider the spatial data introduced in Section 5.2 and a spatial query that requires to find the nearest main towns to *Venice* in any direction. This is an example of direction-based spatial skyline query. The result of this query is the set of towns: $\{Treviso, Padova, Rovigo\}$, since *Treviso* is the nearest main town in the North, *Padova* in the West, and *Rovigo* in the South direction. ◇

5.5.2 Spatial Skyline Query Processing Algorithms

Spatial skyline queries are more challenging than classical skyline queries since the dominance check requires the computation of a derived distance attribute. Properties of the geometric space have therefore to be taken into account in order to define optimal processing solutions for such queries.

Based on what stated in Section 5.5.1, solutions proposed in [41] for dynamic skylines, relying on the usage of R-trees, can still be used to compute spatial skylines. However, as remarked in [51], such solutions have been proposed in order to efficiently process a generic dynamic skyline query. Therefore, they are not tailored to the spatial domain. This is why new processing approaches have been proposed in order to get benefits from the properties of the spatial domain to compute the

Table 5.6 A summary of existing spatial query processing approaches for spatial skyline queries

	Dominance Check	Data Structure for Visiting Data
B^2S^2 [51]	Relationships between skyline points and convex hull of query points	R-Tree for data points in P
VS^2 [51]	Relationships between skyline points, convex hull of query points, and Voronoi cells	Delaunay graph of data points in P
EC [54]	Dominance with respect to the skyline points found so far	L_P list of data points sorted in ascending distance from a query point $q \in CH_v(Q)$
DS [26]	(i) Dominance restricted to adjacent points, with respect to their circular list sorted by the order of directions (ii) Conditions for early termination, based on the angles between adjacent points	R-Tree for data points in P , ordered visit with respect to the distance from q

skyline. Such new approaches rely on a specific distance function, namely the Euclidean distance, and on a branch-and-bound approach applied upon an index tree (e.g., R-tree) or a graph (Delaunay graph) visit. Pruning conditions correspond to the dominance check, in the sense that regions of space that for sure cannot contain skyline points are not visited. Different properties of the spatial domain can be taken into account for this purpose. In the following, the proposed approaches will be presented with respect to the considered data structure to be visited during the computation and the dominance check used for pruning. Table 5.6 summarizes the properties of the described algorithms.

B^2S^2 : Branch-and-Bound Spatial Skyline Algorithm. The branch and bound algorithm for spatial skyline proposed in [51] relies on a typical branch and bound approach like the BB2 scheme presented in Section 5.4.2 to search input data points, indexed by an R-Tree R , avoiding the visit of those subtrees that cannot contain any skyline points. To this purpose, an heap H is maintained, containing the entries to be visited (i.e., potentially containing skyline points), initialized with the R root. Skyline points are collected into a set $S(Q)$. At each step, an entry in H is considered. If it is a leaf entry, it is checked for dominance with respect to the skyline points found so far. In case it is not dominated by any point in $S(Q)$, it is inserted in $S(Q)$ since it corresponds to a new skyline object. If it is an intermediate entry, dominance check is first applied to the entry. In case it is not dominated by any point in $S(Q)$, the dominance check is applied to entries of the pointed node. Each entry whose MBR is not dominated by the current skyline points is inserted into H for further visit.

In order to apply the dominance check, spatial properties of dominance are exploited. Indeed, it can be proved that, given a point p , the points that spatially dominate p (called *dominator region*) and those spatially dominated by p (called *dominance region*) corresponds to well defined regions of the space. More precisely, let $C(q_i, p)$ be the circle centered at the query point q_i with radius $D(q_i, p)$. Each point inside the circle $C(q_i, p)$ is closer to q_i than p . The dominator region of p corresponds to $\bigcap_{q_i \in Q} C(q_i, p)$ while the dominance region corresponds to $\bigcap_{q_i \in Q} \overline{C}(q_i, p)$, where $\overline{C}(q_i, p)$ represents the set of points which are outside $C(q_i, p)$. Finally, the

union of all $C(q_i, p)$ represents the *search region*, namely the set of points that are not spatially dominated by p . Of course, skyline points in P are the points which are not inside the dominance region of any other point.

Based on the previous properties, it is simple to show that, given an entry e , e may lead to detect some skyline points only if e intersects the intersection of the (MBRs of the) search regions of skyline points found so far (called *condition C1* in the following). Thus, dominance check for leaf or intermediate entries (before accessing the entries of the pointed node) is performed only if such condition is satisfied.

Checking dominance using dominance regions is quite expensive, especially when the number of query points is high. To reduce the check cost, sufficient conditions based on the properties between the convex hull of Q and skyline points are provided⁸. In particular, it can be shown that:

- any point $p \in P$ that is inside the convex hull of Q is a skyline point;
- the set of skyline points $S \subseteq P$ does not depend on any point $q \in Q$ that is not a vertex of $CH(Q)$ (called non-convex point).

Based on such properties, if an entry e is completely inside the convex hull $CH(Q)$ (called *condition C2* in the following), e cannot be dominated and therefore it must be inserted into H , if it is an intermediate entry, or it must be inserted into $S(Q)$ if it is a leaf entry. If condition C2 is not satisfied, dominance regions of current skyline points in $S(Q)$ are used to check whether e does not dominate points in $S(Q)$ (called *condition C3* in the following).

The previous properties are also used to define an heuristics for ordering entries in H . In particular, entries are ordered with respect to the sum of their minimum distance to points in $CH_v(Q)$, where $CH_v(Q)$ represents the set of vertexes of $CH(Q)$.

VS²: Voronoi-Based Spatial Skyline Algorithm. An efficient alternative method for computing a spatial skyline consists in traversing the set of data points P using their *Delaunay graph*, instead of an R-Tree [51]⁹. The visit starts from a definite skyline point (e.g., the closest point to one query point, based on Property I), and proceeds from one point to its Voronoi neighbors. Also in this case, an heap H is maintained, containing the points to be further traversed. The heap H is ordered with respect to the sum of the distances of each point with respect to the convex vertices of $CH(Q)$. A minheap HS is also maintained, containing points that are not dominated at the time of the visit but that can be dominated by some points further visited.

⁸ The *convex hull* $CH(Q)$ of a set of points Q is the unique smallest convex polytope that contains all points in Q .

⁹ The *Voronoi cell* of a point $p \in P$ includes all points having p as the closest point, according to some distance metric $D(\cdot)$. In \mathcal{R}^2 , the Voronoi cell is a convex polygon. Each edge of this polygon is a segment of the perpendicular bisector line of the line segment connecting p to another point $p' \in P$ (neighbor of p). The graph having P as vertices and all segments connecting a point to its neighbors as edges is called *Delaunay graph*.

For each visited point p in H , if it is a skyline point, it is inserted in the result set $S(Q)$. Sufficient conditions are used to detect skyline points, based on properties of convex hulls and Voronoi cells, namely:

- Any point $p \in P$ that is inside the convex hull of Q is a skyline point.
- If the interior of the Voronoi cell $VC(p)$ intersects the boundary of $CH(Q)$, p is a skyline point.

If the previous conditions are not satisfied, two other situations may arise: (1) point p is dominated by at least one point in $S(Q) \cup HS$: in this case, p can be discarded and its Voronoi neighbors have not to be visited; (2) point p is not dominated by points in $S(Q) \cup HS$, and therefore it is inserted into HS , since it is not dominated at the time of its examination but it might be dominated later.

If p is not discarded, each Voronoi neighbor of p is considered and, if its Voronoi cells is not dominated by objects in $S(Q)$ and HS , it is inserted in H since it may lead to the identification of further skyline points. A Voronoi cell $VC(p)$ is spatially dominated by a set of points A if and only if it is completely inside the union of the dominance regions of all points in A . An heuristic approach is used for checking cell dominance and reducing the complexity of the computation. As soon as H becomes empty, HS is visited and points of HS that are not dominated by any point in $S(Q)$ are skyline points to be inserted into $S(Q)$.

We notice that HS and the final post processing are required to guarantee the correctness of the algorithm; they were not considered in the preliminary version of VS^2 [50], which was later proved to be incorrect [54]. It has been proved that VS^2 outperforms B^2S^2 .

ES: Enhanced Spatial Skyline Algorithm. In [54] the authors propose an alternative efficient algorithm to VS^2 [50], overcoming the problems identified in the first definition of such algorithm [50]. The proposed algorithm still relies on the usage of Voronoi cells but, instead of traversing the Delaunay graph, the number of dominance checks is reduced by keeping a sorted list L_P of all the data points in the ascending order of their distance from some given vertex of $CH(Q)$ (which, by definition of convex hull, is a query point). For each point in L_P , the algorithm performs the dominance test with respect to the skyline points found so far. The correctness of this algorithm is based on the observation that, if a data point p_1 is located before p_2 in L_P , then p_2 does not spatially dominate p_1 . Therefore, it is sufficient to perform the dominance test on p only with respect to the spatial skyline points that are located before p in L_P . This algorithm can be further improved by computing the *seed skyline points* at the beginning. A seed skyline point is a skyline point that can be identified without dominance test, by checking the properties already considered in B^2S^2 and VS^2 . It has been proved that ES outperforms VS^2 .

DS: Direction-based Spatial Skyline Algorithm. The branch and bound algorithm presented in [26] starts from a query point q and a tolerance value θ for determining when two points are in the same direction. The data points P are indexed with an R-Tree, which is used to explore them in order of their proximity to q , starting from the nearest point to q . Visited points are inserted into a direction list l , sorted by

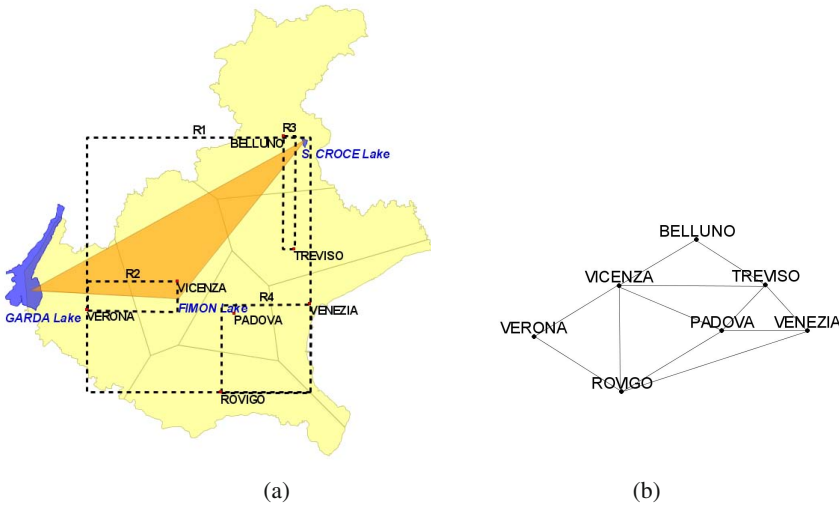


Fig. 5.3 (a) Main towns and lakes in Veneto. Assuming the center of each lake as a query point, the figure also shows: (i) convex hull of lake points; (ii) Voronoy cells; (iii) MBRs constructed upon main towns. (b) Delaunay graph of main towns.

the order of directions with respect to q . Given the i -th nearest point p_i , in order to determine if it is a skyline point, it is sufficient to consider for dominance only its adjacent points in the circular list sorted by the order of directions. Indeed, based on θ , these are the only points that may have the same direction of p . If they do not have the same direction than p (i.e., the angle between each of them and p is greater than θ), p is a skyline object. Otherwise, p can be discarded because there are other points previously visited (namely, nearer to q) in the same direction. An early termination condition, based on the angle between adjacent points, is also provided, ensuring that all objects visited in the future steps of the algorithms are dominated by objects visited so far.

Example 8. In order to illustrate skyline query processing, consider the scenario presented in Section 5.2 and the following spatial skyline query: “Retrieve the main towns that are at the minimum distance from lakes Garda, Fimon, and S. Croce”. In the following, we discuss how the three algorithms B^2S^2 , VS^2 , and ES work. Figure 5.3(a) shows main towns and lakes, the convex hull of query points (one for the center of each lake), Voronoy cells, and MBRs constructed upon main towns (R1 is father of R2, R3, and R4 in the corresponding R-tree). Table 5.7 shows the distances of each main town/MBR with respect to each considered lake.

B^2S^2 . The various steps of the usage of algorithm B^2S^2 on the considered query are presented in Table 5.8. Horizontal lines separate computations performed on the current top entry of H (in bold), say e . If e is a point and it is not dominated by any point in $S(Q)$, it is inserted into $S(Q)$ (since it is a skyline point). If e is an intermediate entry, its entries are visited and, in case they may lead to the

Table 5.7 Distances between main towns/MBRs and considered lakes. Columns d_G , d_F , and d_{SC} contain the distance to the *Garda*, *Fimon*, and *S.Croce* lakes, respectively.

Main Town/MBR	d_G	d_F	d_{SC}	Sum
R_2	27	0	88	115
R_3	120	55	4	179
R_4	89	22	75	395
<i>Vicenza</i>	68	89	8	165
<i>Verona</i>	27	128	43	198
<i>Padona</i>	95	87	27	209
<i>Treviso</i>	124	50	59	233
<i>Belluno</i>	140	11	91	242
<i>Venezia</i>	130	75	61	266
<i>Rovigo</i>	100	123	48	271

Table 5.8 Execution of Algorithm B^2S^2 on a spatial skyline query. Conditions C1, C2, C3 are evaluated on $H.top$ or on its entries, the corresponding columns point out their truth value (true (T) or false (F)). $C1 \wedge (C2 \vee C3)$ has to be satisfied for an entry not to be discarded.

H	$H.top$ entries	C1	C2	C3	$S(Q)$
$[R_1]$		T	F	T	\emptyset
$[\]$	R_2	T	F	T	\emptyset
$[R_2]$	R_3	T	F	T	\emptyset
$[R_2, R_3]$	R_4	T	F	T	\emptyset
$[R_2, R_3, R_4]$		T	F	T	\emptyset
$[R_3, R_4]$	$pVerona$	T	F	T	\emptyset
$[R_3, pVerona, R_4]$	$pVicenza$	T	F	T	\emptyset
$[pVicenza, R_3, pVerona, R_4]$		T	T	T	$\{pVicenza\}$
$[R_3, pVerona, R_4]$		T	F	T	$\{pVicenza\}$
$[pVerona, R_4]$	$pBelluno$	T	F	T	$\{pVicenza\}$
$[pVerona, pBelluno, R_4]$	$pTreviso$	T	F	T	$\{pVicenza\}$
$[pVerona, pTreviso, pBelluno, R_4]$		T	F	T	$\{pVicenza, pVerona\}$
$[pTreviso, pBelluno, R_4]$		T	F	T	$\{pVicenza, pVerona, pTreviso\}$
$[pBelluno, R_4]$		T	F	T	$\{pVicenza, pVerona, pTreviso, pBelluno\}$
$[R_4]$		T	F	T	$\{pVicenza, pVerona, pTreviso, pBelluno\}$
$[\]$	$pPadova$	T	F	T	$\{pVicenza, pVerona, pTreviso, pBelluno\}$
$[pPadova]$	$pVenezia$	T	F	F	$\{pVicenza, pVerona, pTreviso, pBelluno\}$
$[pPadova]$	$pRovigo$	T	F	F	$\{pVicenza, pVerona, pTreviso, pBelluno\}$
$[pPadova]$		T	F	T	$\{pVicenza, pVerona, pTreviso, pBelluno, pPadova\}$

identification of skyline points, (i.e., condition $C1 \wedge (C2 \vee C3)$ is satisfied, where C1, C2, and C3 have been introduced when describing the B^2S^2 algorithm) they are inserted in H .

VS². Under VS^2 , main towns are visited according to the Delaunay graph, presented in Figure 5.3(b), starting from a skyline point. Vicenza is certainly a skyline point, since it is contained in the convex hull of the query points, therefore it is inserted into H and in $S(Q)$. Then, Vicenza is removed from H and replaced by its

neighborhood main towns, ordered with respect to the sum of their distances from vertices of $CH(Q)$, obtaining $H = \{Verona, Padova, Treviso, Belluno, Rovigo\}$ (see Table 5.7). The algorithm then de-heaps Verona from H . Since the interior of its Voronoy cell intersects the boundary of $CH(Q)$, Verona is a skyline point and it is inserted into $S(Q)$. No other main towns are inserted since its neighbor towns are already contained in H . Then, Padova is considered. Padova does not satisfy the sufficient conditions for skyline points; at the same time, its Voronoy cell is not inside the union of the dominance regions of all the other towns, therefore it cannot be discarded but it is inserted into HS . Its neighbor main town Venezia is inserted into H , obtaining $H = \{Treviso, Belluno, Venezia, Rovigo\}$. All the towns have now been inserted into H . Then, Treviso is considered and inserted into $S(Q)$ since it is a skyline point (the interior of its Voronoy cell intersects the boundary of $CH(Q)$). The same situation holds for Belluno. When considering Venezia, it does not satisfy the sufficient conditions for skyline points; at the same time, its Voronoy cell is not inside the union of the dominance regions of all the other towns, therefore it cannot be discarded but it has to be inserted into HS . A similar situation holds for Rovigo. Since H has been completely visited, points in HS are taken into account and their dominance with respect to points in $S(Q)$ is checked. It is simple to show that Belluno is not dominated by points in $S(Q)$ and it can be inserted into $S(Q)$ while Venezia is dominated by Treviso and Rovigo is dominated by Vicenza.

ES. Under algorithm ES, main towns are ordered in an increasing way with respect to their distance to a vertex of $CH(Q)$, for example, Santa Croce lake. We obtain the list: $L = [Vicenza(8), Padova(27), Verona(43), Rovigo(48), Treviso(59), Venezia(61), Belluno(91)]$. It is sufficient to check each main town in the order of the list for dominance with respect to towns that precede it in the list. By Property I, Vicenza is a skyline point. Padova, Verona, Treviso, and Belluno are skyline points since they are not dominated by points preceding them in the list. Rovigo is not a skyline point since it is dominated by Vicenza; Venezia is not a skyline point since it is dominated by Treviso. \diamond

5.6 Approximate Query Processing

Approximate query processing refers to all the techniques for executing an intrinsically expensive query by using ad hoc query processing algorithms that automatically apply the minimum amount of relaxation based on the available data and resources, in order to efficiently compute a non-empty result close to the user request. These techniques are applied in those situations where limited resources do not allow to produce an exact answer in a short time or where data can be quite heterogeneous and may contain errors.

As pointed out in Section 5.3, independently on the considered data model, ApQP approaches can be classified into four groups, corresponding to: (i) relaxation of existing query processing algorithms; (ii) definition of new algorithms over the input

datasets; (iii) usage of heuristics in exploring the solution space; (iv) approximation of the input datasets and design of ad hoc execution algorithms for the new data. As a result, differently from the techniques presented in Sections 5.4 and 5.5, ApQP techniques proposed so far are very heterogeneous and it is not possible to find additional common aspects beside those leading to the proposed first-level classification. In order to give the reader a feeling of how such algorithms look like, in the following we present three specific approaches:

- ApQP techniques for the multiway spatial join operation, relying on heuristics in exploring the search space in order to more quickly reach a good approximate result [39] (approach (iii));
- ApQP techniques for k NN selection and join queries, based on the relaxation of already existing algorithms for NN query execution [16, 17, 20] (approach (i));
- ApQP techniques based on the approximation of input data (approach (iv) and (ii)); as an example, we illustrate the approach presented in [3, 62], approximating each spatial object with a raster signature.

5.6.1 Approximate Algorithms for Multiway Spatial Join

A multiway spatial join is a sequence of join operations involving three or more datasets where the join condition relies on a spatial predicate. Formally, a multiway spatial join can be defined as follows: given n datasets D_1, \dots, D_n and m join conditions $Q = \{Q_{i,j}\}$, referring to datasets D_i and D_j , the multiway spatial join $\bowtie_Q(D_1, \dots, D_n)$ is the set of n -tuples $\{(t_{1,w}, \dots, t_{i,x}, \dots, t_{j,y}, \dots, t_{n,z}) \mid \forall i, j, t_{i,x} \in D_i, t_{j,y} \in D_j, \text{ and } t_{i,x} Q_{i,j} t_{j,y}\}$. An example of a three-way spatial join, referring to our running example presented Section 5.2, is the following: “Find all the triples (m, l, r) where the main town territory m contains the lake l and also crosses the river r ”.

Several approaches have been proposed for the processing of the multiway spatial join in an exact way. However, as far as we know, only few approaches exist for the approximate version of this problem, whose relevance is motivated by the high computational complexity of multiway spatial join, which is in general exponential. One of them has been proposed in [39] and considers only *intersects* as query predicate (two spatial objects satisfy the *intersects* relation if they are not disjoint). As shown in [36], a multiway spatial join operation can return an approximate result by introducing the concept of *inconsistency degree* of a tuple, corresponding to the number of join conditions that the tuple does not satisfy. When the multiway spatial join is executed in a precise way, only the tuples with *zero inconsistency degree* are returned as result. This behavior can be approximated by returning also tuples with low *inconsistency degree*, thus reducing the number of join conditions that must be satisfied.

Example 9. Consider the data shown in Figure 5.4, where main towns are represented as polygons, and the query introduced above. The inconsistency degree of tuple $\langle \text{Mantova}, \text{Superiore lake}, \text{Mincio} \rangle$ is 0 since Mantova contains Superiore lake and crosses the Mincio river. On the other hand, the inconsistency degree of

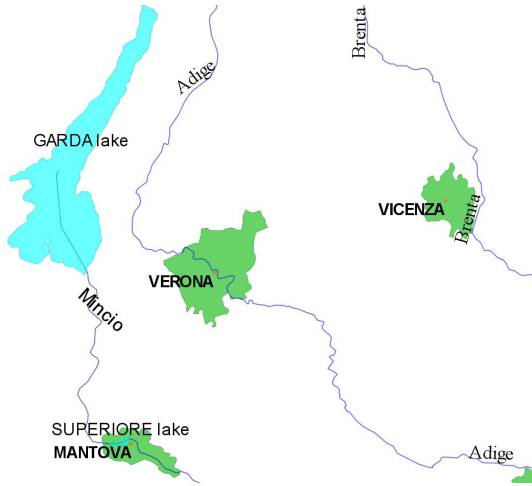


Fig. 5.4 Example of data that produce tuple with different inconsistency degrees.

tuple $\langle Verona, Garda\ lake, Adige \rangle$ is 1 since Verona crosses the Adige River but does not contain the Garda lake; tuple $\langle Vicenza, Garda\ lake, Brenta \rangle$ has 2 as inconsistency degree since Vicenza neither contains the Garda lake nor crosses the Brenta river. \diamond

Concerning the implementation of approximate multiway spatial join algorithms, the authors propose three different approaches, *Indexed Local Search*, *Guided Indexed Local Search*, and *Spatial Evolutionary Algorithm*, illustrated in the following. They all represent the search space as a graph where each solution (a tuple t of n geometries) corresponds to a node having some inconsistency degree with respect to the join condition. The number of nodes in the graph depends on the datasets cardinality and is equal to: $|D_1| \times \dots \times |D_n|$ if the join involves n datasets. Edges connect nodes that are at distance 1, i.e., two connected nodes represent tuples that differ only in one geometry. Thus, each node is connected to n other nodes.

Indexed Local Search (ILS): ILS searches for a *local maximum* in the graph starting from a random solution called *seed*. A local maximum represents a node (i.e., a tuple) that has lower inconsistency degree with respect to all its neighbors; when a solution is found, the algorithm restarts with a different seed until a fixed timeout is reached. The technique uses R*-trees to improve performance.

Guided Indexed Local Search (GILS): GILS improves ILS by introducing some “memory” mechanism in order to avoid that the algorithm may find the same local minimum more times; in particular, GILS keeps a list of geometries that have participated to some already found local maximum and it assigns them a penalty. Given a new local maximum, the penalty is assigned to those geometries with the minimum penalty so far in order to avoid over-punishing. The penalty is then used

for computing a new inconsistency degree, called *effective inconsistency degree*, which is obtained by adding the penalties to the actual inconsistency degree.

Spatial Evolutionary Algorithm (SEA): the evolutionary approach is based on the concept of natural mutation and survival of the fittest individuals. After instantiating an initial population of solutions P , three genetic operations are applied: *selection*, *crossover*, and *mutation*. The first operation evaluates the similarity of the solutions in P (*evaluation*), then each solution s is compared with a set of t random solutions $\{s_1, \dots, s_t\}$ and s is replaced with the best among $\{s, s_1, \dots, s_t\}$ (*tournament*). The second operation applies a combination among the available solutions in P (*crossover mechanism*). In particular, pairs of solutions (s_1, s_2) are selected randomly from P , then a certain number of geometries are preserved (initially only one, afterward an increasing number) and the other ones are mutually exchanged. Finally, the *mutation* operation is applied to each solution $s \in P$; it then modifies s with the same algorithm used for ILS. These operations are applied in sequence and iteratively until a timeout is reached.

Experiments presented in [39] show that SEA significantly outperforms both ILS and GILS with respect to both performance and accuracy of the obtained result.

5.6.2 Approximate Algorithms for Distance-Based Queries

Distance-based queries represent another relevant group of spatial operations for which approximate algorithms have been proposed. Besides NN selection and join, distance-based queries include, among the others, k NN selection and k NN join (a complete list can be found in [19]). k NN queries extend NN selection and join, presented in Section 5.2, to return the k objects closest to the query object (k NN selection) or, given two datasets, for each object in the first dataset, its k nearest neighbors in the second dataset (k NN join) are detected. While algorithms for k NN queries guarantee good performance for low dimensional spaces, their performance degrades as the number of dimensions increases.

According to [19], existing approximate solutions for executing k NN selection and join belong to two distinct groups. Techniques in the first group modify traditional k NN algorithms by reducing the search space through the usage of ad hoc pruning conditions and stopping criteria. Thus, they belong to group (i) according to our classification. The second group coincides with group (iii) according to the proposed classification. While no specific heuristic-based proposals for distance-based queries have been presented, as discussed in [19], typical algorithm schemes proposed for other operators (e.g., multiway spatial join [39]) can however be easily adapted to the purpose.

Specific approaches belonging to the first group have been presented in [20], for distance-based selection and join, and in [17] for distance-based multiway spatial join. They all assume that spatial data are indexed through a tree of the R-tree family. Then, a typical R-tree based branch-and-bound visit (like the ones described in Section 5.4.2) is modified by reducing the search space as follows:

- A depth-first traversal is used, thus giving higher priority to the closest R-tree nodes and updating the pruning distance very quickly. In this way, acceptable approximate solutions are usually available when the processing of the algorithm is stopped before its normal termination, as occurs in the approximate case.
- Some heuristics are introduced in the pruning step, in order to avoid the visit of some parts of the tree if it is highly probable that they will not contain any answer.

Various pruning conditions have been proposed. Among them, we recall the following:

- *α -allowance methods*: the pruning heuristic is strengthened by discarding an (intermediate or leaf) entry x when $MINMINDIST(x,y) + \alpha(z) > z$. Here, $MINMINDIST(x,y)$ is the function that computes the distance between x and the query point (for selection) or another dataset point y (for join); z is the distance of the k -th closest point (or pair) found so far. $\alpha(z)$ may return a non negative constant β or $\gamma \times z$ where γ is a constant in the interval $[0, 1]$.
- *N -consider*: the internal node visiting step is modified by introducing a maximal percentage N ($0 \leq N \leq 1$) of items to be considered inside each node. All items above the N percentage are discarded and node visit terminated.
- *M -consider*: the algorithm terminates when a specified percentage M ($0 \leq M \leq 1$) over the total number of items examined by the exact algorithm is reached. Of course the number of items examined by the exact solution cannot be computed, but it is estimated on the basis of the dataset cardinality, distribution and dimensionality and stored in a look-up table.

Experimental results show that the *N -consider* method exhibits the best performance and it is recommended when the users are more interested in a good response time at the price of a lower accuracy; *α -allowance* guarantees instead an opposite behavior.

Example 10. In order to illustrate the effect of the α -allowance method in the generation of approximate results, consider the k NN selection query that, given a lake (specifically the Corlo lake) finds the k NN main towns (assuming they are represented as polygons) of Northern Italy (see Figure 5.5). In the example the applied approximation function is $\alpha(z) = 0.1 * z$. The α -allowance method results in pruning subtree $R2$ since it satisfies the condition $MINMINDIST(R2, Corlo\ lake) + \alpha(z) > z$. Notice that, without introducing the $\alpha(z)$ function, the subtree $R2$ would have been considered in the execution of the k NN selection, since it does not satisfy the condition $MINMINDIST(R2, Corlo\ lake) > z$. \diamond

5.6.3 Algorithms Based on Approximate Spatial Data

The introduction of an approximate representation of the exact geometries is a well-known approach, typically used to filter out non-interesting data in the processing of any spatial query. A refinement step then returns the precise result, identifying all false hits generated by the filter step, at the price of executing potentially expensive spatial operations over the precise geometry. As an example, R-trees and

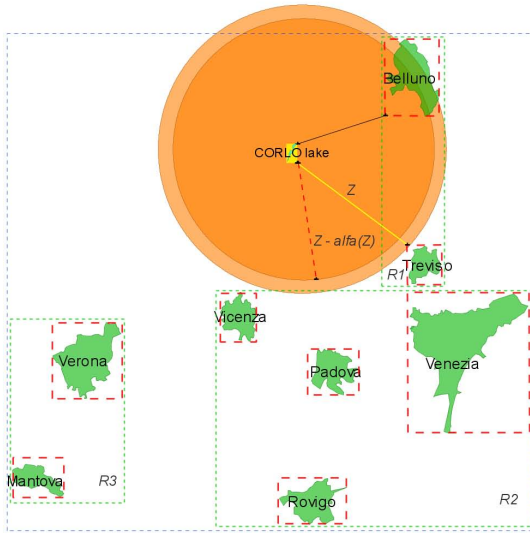


Fig. 5.5 Example of application of the α -allowance methods to the execution of a k NN selection query that retrieves the k closest main towns (considered as polygons).

their variants use the Minimum Bounding Rectangle (MBR) as an approximation of data geometry [27]. A more efficient but approximate solution to the approach described above is to consider the result generated by the filtering step as an “approximate query result”. In case of R-trees, such result is complete (i.e., all results of the precise queries are returned) but not sound (some false hits can be returned).

The MBR is a coarse representation of a geometry and this leads to a low accuracy of the approximate query result. However, other spatial data approximation have been proposed that lead to more efficient approximate solutions. One of such approaches approximate each geometry value with a 4-color raster signature (4CRS) [3, 62]. 4CRS is a compact description of a geometry shape and extension composed of a small bit-map of four colors upon a grid of cells. In particular, each cell has a color representing the percentage of polygon area within the cell: *Empty* (0% of polygon area in the cell), *Weak* (50% or less of polygon area in the cell), *Strong* (more than 50% of polygon area in the cell and less than 100%) and *Full* (the polygon covers the cell). The grid scale can be changed in order to obtain a finer or coarser representation of the polygon geometry.

Specific algorithms have been proposed to execute spatial queries over 4CRSs, including: (i) spatial operators that return some kind of geometry measure, like area, distance, diameter, length, perimeter, number of components; (ii) spatial selection and join with respect to various predicates (equal, different, disjoint, inside, area disjoint, edge disjoint, edge inside, vertex inside, intersects, meet, adjacent, border in common); (iii) set operators, like intersection, minus and other operators returning geometries like common border, vertices, contour, interior; (iv) other operations

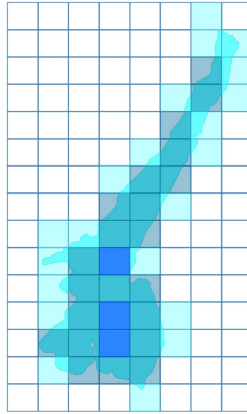


Fig. 5.6 Example of 4CRS representation of real data (Garda lake).

on object sets, like sum, closed, decompose, overlay, fusion. The idea is always to estimate the operation result, when applied to the exact geometries, by means of a simplified algorithm applied to the corresponding 4CRSs.

Each spatial predicate is approximated by a function that returns a value between 0 and 1 that indicates the true percentage of the spatial predicate. The function behavior depends on the predicate type. As an example, for checking equality, the cells of the object MBRs are compared and a value between 0 and 1, called *affinity degree*, is assigned to each pair of corresponding cells. The affinity degree is 1 when the cells are both *Empty* or both *Full*; in all the other cases, it is equal to the expected intersection area: for instance, the pair *Weak* \times *Weak* contributes with 0.0625, while the pair *Strong* \times *Strong* with 0.5625. Of course, if the type of two corresponding cells is different, the objects are not equal. As example of 4CRS, Figure 5.6 shows the 4CRS of the Garda lake.

Experimental results show that the approximate algorithms are from 3.5 to 14 times faster than the exact processing in response time while the response error is at most the 3% of the precise result.

5.7 Towards Qualitative Approximation Techniques for Spatial Data

As we have seen in the previous sections, most of the approaches proposed for preference-based spatial queries rely on the usage of some ranking function based on quantitative, i.e., metric (distance-based), information. In *top-k* queries, such function is used for ranking the result, in skyline queries, it is used to define dominance. However, as pointed out in Table 5.1, only few approaches approximate queries taking into account qualitative information, for example represented by topological and cardinal spatial relations. Qualitative spatial relations compare pairs of spatial

objects based on their mutual positions in the space and are frequently used to specify spatial, not approximate, queries in real applications (see Section 5.2).

Based on this consideration, in the following, for each type of preference-based query described in this chapter (namely, top- k and skyline), we present a new version of the query that, instead of using some kind of metric distance, is based on spatial qualitative relations. Moreover, we discuss how the query processing algorithms already defined for top- k and skyline queries based on metric distance can be extended to cope with them. In order to do that, we first introduce an approach for dealing with qualitative spatial relations in a quantitative way. For the sake of simplicity, we focus on spatial selection but the provided definitions can be easily extended to spatial join or other spatial operations.

5.7.1 From Qualitative to Quantitative Spatial Relations

The simplest way to quantify the difference between two qualitative spatial relations is to rely on a distance function, in order to convert the qualitative difference into a number. Given a set of qualitative spatial relations \mathcal{G} , for example topological relations, a distance function $D_{\mathcal{G}}$ applied on two spatial relations θ_1 and θ_2 in \mathcal{G} returns a value between 0 and 1 quantifying the difference between θ_1 and θ_2 . A value equal to zero means that θ_1 and θ_2 coincide.

In the literature, several distance functions for topological and cardinal relations have been proposed. Concerning topological relationships, some initial proposals deal with conceptual neighbor graphs for topological relations [21], which define a partial order among specific sets of topological relations (e.g., surface-surface relations).

In [11], topology distance is used to evaluate similarity of spatial scenes, by taking into account also direction and distance relations. In [23], topology distance is used to define a model (snapshot model) to compare two different topological relations between lines and surfaces. In all the papers cited above, similarity is computed only between pairs of objects with the same dimension. Multiple object representations are not considered at all. More recent proposals extend the distance functions proposed in [21] (only for surfaces) and in [23] (for lines and surfaces) to geometry type-independent set of topological or cardinal relations, computing a value between 0 and 1 based on the matrix representation of the considered relations [6, 7, 44].

Distance functions for cardinal relations have been proposed in [7, 25]. In [25], two distance functions for cardinal relations have been defined. The first is defined for single-tile relations, i.e., relations corresponding to intersections of the target object with a single-tile (see Section 5.2), and it corresponds to the minimum length of the paths connecting the two directions in a *conceptual graph*. Such graph contains a node for each tile and one edge between pairs of tiles sharing at least one border. The second is defined for multi-tile relations, i.e., relations corresponding to intersections of the target object with multiple tiles, and it considers the percentage of target object belonging to each tile. The main problem of this second approach

is that it does not rely on the model used for defining cardinal relations. Another distance function overcoming this problem has been proposed in [7].

Based on the chosen distance function $D_{\mathcal{G}}$, it is possible to define a distance function $d_{\mathcal{G}}^{\theta}$ between pairs of spatial objects, with respect to a query relation $\theta \in \mathcal{G}$, as follows.

Definition 8 (\mathcal{G} -based spatial distance). Let \mathcal{G} be a set of qualitative spatial relations. Let $\theta \in \mathcal{G}$. Let f and g be two spatial objects such that $f\theta'g$ holds, $\theta' \in \mathcal{G}$. Let $D_{\mathcal{G}}$ be a distance function for \mathcal{G} . The \mathcal{G} -based spatial distance between f and g , based on $D_{\mathcal{G}}$ with respect to θ , is defined as follows: $d_{\mathcal{G}}^{\theta}(f, g) = D_{\mathcal{G}}(\theta, \theta')$ \square

Example 11. Consider the set of topological relations defined for surfaces $\mathcal{G} = \{\text{disjoint}(d), \text{touches}(t), \text{overlaps}(o), \text{within}(i), \text{contains}(c), \text{equals}(e), \text{coveredBy}(b), \text{covers}(v)\}$ where *coveredBy* (*covers*) is like *within* (*contains*) with touching boundaries, while *within* and *contains* require non touching boundaries.

Let $D_{\mathcal{G}}(\theta_1, \theta_2)$ be defined upon the conceptual neighbor graph, shown in Figure 5.7 and taken from [21], as the sum of the weight of the edges composing the shortest path between θ_1 and θ_2 in the graph. For example, $D_{\mathcal{G}}(d, d) = 0$, $D_{\mathcal{G}}(d, t) = 1$ and $D_{\mathcal{G}}(d, e) = 10$.

The \mathcal{G} -based spatial distance between f and g , based on $D_{\mathcal{G}}$ defined as above with respect to *disjoint*, namely, $d_{\mathcal{G}}^d(f, g)$, is equal to zero if f and g are *disjoint*, otherwise it has a value that measures the similarity between *disjoint* and the existing relation θ' between f and g . For example, if f is contained into g , then $d_{\mathcal{G}}^d(f, g) = D_{\mathcal{G}}(d, i) = 8$. \diamond

In general, a \mathcal{G} -based spatial distance is not symmetric and the triangular inequality does not hold for it. Additionally, there is no relationship between the space, the objects which are embedded in, and the values returned by the distance function. As a consequence, we cannot exploit these concepts for specializing or optimizing the processing algorithms that have been presented in Sections 5.4 and 5.5 as it will be explained in Section 5.7.2.

Property 2. A \mathcal{G} -based spatial distance is in general not symmetric and the triangular inequality does not hold for it.

Proof. Consider the set of topological relations and the \mathcal{G} -based spatial distance function presented in Example 11. Such function is not symmetric. Indeed, given f and g where f contains g , then $d_{\mathcal{G}}^c(f, g) = 0$, while $d_{\mathcal{G}}^c(g, f) = 8$, since g in f . In order to show that the triangular inequality is not satisfied, consider two objects f and g such that f in g and, as consequence, $d_{\mathcal{G}}^t(f, g) = 7$. Let now consider a third object h such that f disjoint h and h disjoint g , then $d_{\mathcal{G}}^t(f, h) = 1$ and $d_{\mathcal{G}}^t(h, g) = 1$, thus the sum of the two distances is 2 that is less then 7 and the triangular inequality is not satisfied. \square

¹⁰ In the following, we omit the reference to $D_{\mathcal{G}}$ and θ in presenting a \mathcal{G} -based spatial distance function when they are clear from the context.

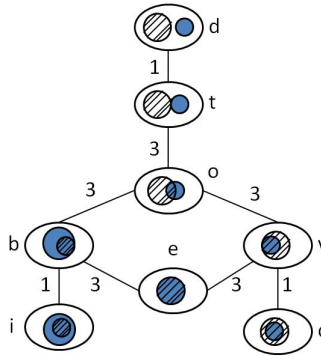


Fig. 5.7 Conceptual graph for topological relationships among surfaces [21].

5.7.2 Spatial Top- k Queries Based on Qualitative Relations

The qualitative spatial distance presented in Section 5.7.1 can be used to define top- k queries based on relations in \mathcal{G} . Examples of queries we may want to answer, with respect to the scenario introduced in Section 5.2, are the following:

- Assuming that main towns and rivers are both represented as polygons, find the top-100 main towns that overlap river *Piave*, using a topological distance function $d_{\mathcal{G}}^o(o, Piave)$ to quantify the score between each object o and river *Piave* with respect to *overlaps*. In the result list, we expect municipalities that effectively overlap *Piave* be listed before, followed by municipalities that satisfy a topological relation close to *overlaps*, in the order of their score, according to the considered distance function (e.g., those touching or covering or are covered by the river, according to the distance function induced by the graph presented in Figure 5.7).
- Find the top-2 municipalities in Veneto that are located on the South of *Vicenza*, using a cardinal distance function $d_{\mathcal{G}}^{South}(o, Vicenza)$ to quantify the score between each object o and *Vicenza* with respect to *South*. In the result list, we expect *Rovigo* be listed first, since it is located on the South of *Vicenza*, followed by one between *Verona* and *Padova*, which represent points that are on the South or on the West and East of *Vicenza*, respectively.

The previous queries can be defined as top- k queries with respect to the ranking function introduced by the following definition.

Definition 9 (\mathcal{G} -based ranking function). Let f be a spatial object and q a query object. Let $d_{\mathcal{G}}^{\theta}$ be a \mathcal{G} -based spatial distance. A \mathcal{G} -based ranking function, on $d_{\mathcal{G}}^{\theta}$, can be defined as follows:

$$\tau_q^{\theta}(f) = \mathcal{F}(d_{\mathcal{G}}^{\theta}(f, q))$$

where $\mathcal{F}(\dots)$ is a function over distance values. □

The query examples presented above can be interpreted as top- k queries based on a \mathcal{G} -based ranking function, defined upon the \mathcal{G} -based spatial distance introduced in Section 5.7.1 where $\mathcal{F}(\dots)$ is the identity function. Other ranking functions may for example weight a \mathcal{G} -based spatial distance with respect to other spatial information, e.g., Euclidean distance between objects or non-spatial attributes.

We notice that, when $\mathcal{F}(\dots)$ is the identity function, the proposed ranking function just requires the \mathcal{G} -based distance function to compute the score. This means that top- k algorithms introduced in Section 5.4 can be easily extended to compute top- k queries based on the considered \mathcal{G} -based ranking function. More precisely, in case of spatial selection and algorithm scheme BB2 (see Subsection 5.4.2), the visit of an index node is discarded if it can be determined that the subtree rooted by the node does not contain any object satisfying a predicate, which is closer to the query predicate than top- k solutions found so far. To this aim, as discussed in [8], each entry can be associated with a key value $[dmin, dmax]$. Such interval is defined by a lower bound for the minimum distance and an upper bound for the maximum distance between the query relation and the relations satisfied by the query object and the objects contained in the subtree rooted by the entry. The distance range can be easily computed relying on the notion of *compatibility* for topological relations in an R-tree, first presented in [42]. The basic idea behind compatibility is that the relationship between any object indexed by a given entry e and the query object O cannot be arbitrary but must be *compatible* with the relationship existing between the MBR associated with e and the MBR of O .

Of course, the solution just described works if $\mathcal{F}(\dots)$ is the identity function. Other functions may require the usage of additional data structures and the design of new ad hoc query processing algorithms.

Example 12. Consider the set of topological relations and the \mathcal{G} -based spatial distance function presented in Example 11 as well as the set of main towns in Veneto, represented as polygons, indexed as described in Example 4 and shown in Figure 5.2. Suppose you want to compute the top-2 main towns which overlap the *Piave* river (see Figure 5.2). The algorithm proceeds as described in Subsection 5.4.2. Consider the entry e , associated with the MBR R_2 . $key(e)$ corresponds the interval $[dmin, dmax]$, where $dmin$ is a lower bound for the minimum distance and $dmax$ is an upper bound for the maximum distance between the relations satisfied by descendants of R_2 (i.e. R_7 and R_8) with respect to the MBR of *Piave* and the query relation *overlaps*. In order to compute such interval, without accessing the subtree rooted by R_2 , first the topological relation satisfied by R_2 and the MBR of the query object is computed. In this case, it is *disjoint*. Then, based on compatibility relations, the set of relations that descendants of R_2 can satisfy with respect to the query object are deduced according to the rules provided in [42] and minimum and maximum distances are computed. In this case, it is simple to show that the only relation which is compatible with *disjoint* is *disjoint* itself, therefore the interval will be set to $[4, 4]$, since $D_{\mathcal{G}}(d, o) = 4$ according to the distance function presented in Example 11. On the other hand, when considering R_3 , R_3 and the MBR of the query object overlap. Based on [42], relations which are compatible with *overlaps* are *disjoint*, *touches*, *within*, *coveredBy*, *overlaps*, whose distances with respect to *overlaps* are 4, 3, 4, 3, 0, respectively. Thus, $key(e)$ is set to $[0, 4]$. \diamond

5.7.3 Spatial Skyline Queries Based on Qualitative Relations

The qualitative spatial distance presented in Section 5.7.1 can be used to define spatial skyline queries based on relations in \mathcal{G} . Examples of queries we may want to answer, with respect to the scenario introduced in Section 5.2 are the following:

- Find the municipalities that overlap lake *Garda* and are crossed by river *Adige*. We expect to find all municipalities m that either satisfy the given conditions or, if no municipality satisfies both of them, those that satisfy in the best way some similar conditions, according to two topological spatial distance functions $d_{\mathcal{G}}^o$ and $d_{\mathcal{G}}^r$.
- Find the provinces located on the South of *Verona*, using a cardinal distance function $d_{\mathcal{G}}^{South}$ to quantify the score between each province o and *Verona* with respect to *South*. No province precisely satisfies this condition, however, *Rovigo* is the best approximation since it is located on South-East with respect to *Verona*.

The previous queries can be defined as spatial skyline queries, as in Section 5.5 relying on the following notion of dominance.

Definition 10 (\mathcal{G} -based Spatial Dominance). Let P be a set of spatial objects, $Q = \{q_1, \dots, q_n\}$ a list of spatial query objects, and $\Theta = \{\theta_1, \dots, \theta_n\} \subseteq \mathcal{G}$ a set of query relations (thus, the query predicate is: $p\theta_1q_1 \wedge \dots \wedge p\theta_nq_n$ where $p \in P$). Let $d_{\mathcal{G}}^{\theta}$ be a \mathcal{G} -based spatial distance. A spatial object $p \in P$ spatially dominates another object $p' \in P$ if and only if:

- $d_{\mathcal{G}}^{\theta_i}(p, q_i) \leq d_{\mathcal{G}}^{\theta_i}(p', q_i)$ for all $q_i \in Q$, $\theta_i \in \Theta$, and
- $d_{\mathcal{G}}^{\theta_j}(p, q_j) < d_{\mathcal{G}}^{\theta_j}(p', q_j)$ for some $q_j \in Q$ and $\theta_j \in \Theta$. □

The \mathcal{G} -based Spatial Skyline can be defined as the subset of the input objects that are not \mathcal{G} -spatially dominated by any other object in the input dataset, with respect to the query objects, according to a given \mathcal{G} -based spatial distance.

It is simple to show that also \mathcal{G} -based spatial skyline queries, similarly to spatial skyline queries, can be defined as a special case of *dynamic skyline* queries, introduced in [41]. On the other hand, differently from spatial skyline queries, there is no more a clear relation between the space of input and query objects and the domain of attributes to be considered for the skyline computation. Additionally, as discussed in Section 5.7.1 a \mathcal{G} -based distance function is not necessarily symmetric and does not necessarily satisfy the triangular inequality. As a consequence, while techniques proposed in [41] can still be used for processing \mathcal{G} -based spatial skyline queries, specific techniques proposed for spatial skyline queries cannot.

Future work is therefore needed in order to define optimized query processing techniques for \mathcal{G} -based spatial skyline queries. Some preliminary work in this direction has been proposed in [8], where a best-fit query has been introduced, representing a \mathcal{G} -based spatial skyline with respect to a single object. The proposed algorithm explores the data space using an R-tree. Since just one query object is considered, the dominance check corresponds to verify a disequality between two distances. Pruning is performed using the $[dmin, dmax]$ key value discussed in Section 5.7.2. However, the proposed approach cannot be trivially extended to cope with

more than one query object; new processing solutions have therefore to be designed in order to cope with this more general problem.

5.8 Conclusion and Open Issues

Approximation techniques for spatial data have recently moved from data representation to query result issues. As a consequence, query-based approximation techniques previously defined for traditional data have been extended to cope with the spatial domain. In this chapter, we have classified and surveyed query-based approximation techniques for spatial data. Three main groups of approaches have been identified: query rewriting, preference-based queries, and approximate query processing. Most existing approaches for spatial data belong to the last two categories. We have therefore presented in details and further classified preference-based (namely, top- k and skyline queries) and approximate query processing approaches.

Based on the reported analysis, summarized in Table 5.1, two main considerations follow. First of all, most of the proposed approaches for spatial approximate queries rely on the usage of quantitative, i.e., distance-based information. On the other hand, only few of them approximate queries taking into account qualitative information, for example by considering topological and cardinal relations. Based on this consideration, we have provided new types of queries relying on qualitative relations and discussed how the query processing algorithms already defined for metric relations can be extended to cope with them. Future work is however needed to provide optimized approaches for processing such new queries in an efficient and effective way. As a second consideration, as far as we know, no approaches have been proposed so far for shrinking the result of spatial queries, in order to solve the many answer problem. An additional relevant issue to be investigated therefore concerns how solutions already proposed for non-spatial data, like those presented in [33, 37], can be extended to cope with the spatial domain.

References

1. Acharya, S., Gibbons, P.B., Poosala, V., Ramaswamy, S.: Join Synopses for Approximate Query Answering. In: SIGMOD Conference, pp. 275–286 (1999)
2. Agrawal, S., Chaudhuri, S., Das, G., Gionis, A.: Automated Ranking of Database Query Results. In: CIDR (2003)
3. Azevedo, L.G., Zimbrão, G., de Souza, J.M.: Approximate Query Processing in Spatial Databases Using Raster Signatures. In: GeoInfo, pp. 53–72 (2006)
4. Babcock, B., Chaudhuri, S., Das, G.: Dynamic Sample Selection for Approximate Query Processing. In: SIGMOD Conference, pp. 539–550 (2003)
5. Bakalov, P., Hadjieleftheriou, M., Tsotras, V.J.: Time Relaxed Spatiotemporal Trajectory Joins. In: GIS, pp. 182–191 (2005)
6. Belussi, A., Boucelma, O., Catania, B., Lassoued, Y., Podestà, P.: Towards Similarity-Based Topological Query Languages. In: Grust, T., et al. (eds.) EDBT 2006. LNCS, vol. 4254, pp. 675–686. Springer, Heidelberg (2006)

7. Belussi, A., Catania, B., Podestà, P.: Using Qualitative Information in Query Processing over Multiresolution Maps. In: *Spatial Data on the Web: Modeling and Management*, pp. 159–186. Springer, Heidelberg (2007)
8. Belussi, A., Catania, B., Podestà, P.: Topological Operators: a Relaxed Query Processing Approach. *GeoInformatica* 16(1), 67–110 (2012)
9. Berchtold, S., Böhm, C., Keim, D.A., Kriegel, H.P.: A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space. In: *PODS*, pp. 78–86 (1997)
10. Börzsönyi, S., Kossmann, D., Stocker, K.: The Skyline Operator. In: *ICDE*, pp. 421–430 (2001)
11. Bruns, H.T., Egenhofer, M.J.: Similarity of Spatial Scenes. In: *7th Symposium on Spatial Data Handling*, pp. 31–42 (1996)
12. Butenuth, M., Gösseln, G.V.: Integration of Heterogeneous Geospatial Data in a Federated Database. *ISPRS Journal of Photogrammetry and Remote Sensing* 62(5), 328–346 (2007), Theme Issue: Distributed Geoinformatics - From Sensors to Systems, <http://www.sciencedirect.com/science/article/pii/S0924271607000275>, doi:10.1016/j.isprsjprs.2007.04.003
13. Catania, B., Guerrini, G.: Towards Adaptively Approximated Search in Distributed Architectures. In: Vakali, A., Jain, L.C. (eds.) *New Directions in Web Data Management 1. Studies in Computational Intelligence*, vol. 331, pp. 171–212. Springer, Heidelberg (2011)
14. Chakrabarti, K., Garofalakis, M.N., Rastogi, R., Shim, K.: Approximate Query Processing using Wavelets. *VLDB J* 10(2-3), 199–223 (2001)
15. Clementini, E., Felice, P.D., van Oosterom, P.: A Small Set of Formal Topological Relationships Suitable for End-User Interaction. In: *SSD*, pp. 277–295 (1993)
16. Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M.: Closest Pair Queries in Spatial Databases. In: *SIGMOD Conference*, pp. 189–200 (2000)
17. Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M.: Distance Join Queries of Multiple Inputs in Spatial Databases. In: Kalinichenko, L.A., Manthey, R., Thalheim, B., Wloka, U. (eds.) *ADBIS 2003. LNCS*, vol. 2798, pp. 323–338. Springer, Heidelberg (2003)
18. Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M.: Multi-Way Distance Join Queries in Spatial Databases. *GeoInformatica* 8, 373–402 (2004)
19. Corral, A., Vassilakopoulos, M.: Approximate Computation of Distance-Based Queries. In: *Spatial Databases*, pp. 130–154 (2005)
20. Corral, A., Vassilakopoulos, M.: On Approximate Algorithms for Distance-Based Queries using R-trees. *Comput. J.* 48(2), 220–238 (2005)
21. Egenhofer, M.J., Al-Taha, K.K.: Reasoning about Gradual Changes of Topological Relationships. In: *Spatio-Temporal Reasoning*, pp. 196–219 (1992)
22. Egenhofer, M.J., Franzosa, R.D.: Point Set Topological Relations. *Int. Journal of Geographical Information Systems* 5, 161–174 (1991)
23. Egenhofer, M.J., Mark, D.M.: Modeling Conceptual Neighborhoods of Topological Line-Region Relations. *Int. Journal of Geographical Information Systems* 9(5), 555–565 (1995)
24. Goodchild, M.F.: Measurement-based GIS. In: Shi, W., Fisher, P., Goodchild, M. (eds.) *Spatial Data Quality*, pp. 5–17. Taylor and Francis (2002)
25. Goyal, R.K., Egenhofer, M.J.: Consistent Queries over Cardinal Directions across Different Levels of Detail. In: *DEXA Workshop*, pp. 876–880 (2000)
26. Guo, X., Ishikawa, Y., Gao, Y.: Direction-Based Spatial Skylines. In: *MobiDE*, pp. 73–80 (2010)

27. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: SIGMOD Conference, pp. 47–57 (1984)
28. Hjaltason, G.R., Samet, H.: Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.* 24(2), 265–318 (1999)
29. Hsueh, Y.-L., Zimmermann, R., Yang, M.-H.: Approximate Continuous K Nearest Neighbor Queries for Continuous Moving Objects with Pre-Defined Paths. In: Akoka, J., Liddle, S.W., Song, I.-Y., Bertolotto, M., Comyn-Wattiau, I., van den Heuvel, W.-J., Kolp, M., Trujillo, J., Kop, C., Mayr, H.C. (eds.) *ER Workshops 2005*. LNCS, vol. 3770, pp. 270–279. Springer, Heidelberg (2005)
30. Ilyas, I.F., Beskales, G., Soliman, M.A.: A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Comput. Surv.* 40(4), 11:1–11:58 (2008)
31. Inc., O.G.C.: OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture. *OpenGIS Implementation Standard* (2010)
32. Jhingran, A.: Enterprise Information Mashups: Integrating Information, Simply. In: *VLDB*, pp. 3–4 (2006)
33. Kadlag, A., Wanjari, A.V., Freire, J.-L., Haritsa, J.R.: Supporting Exploratory Queries in Databases. In: Lee, Y., Li, J., Whang, K.-Y., Lee, D. (eds.) *DASFAA 2004*. LNCS, vol. 2973, pp. 594–605. Springer, Heidelberg (2004)
34. Kossmann, D., Ramsak, F., Rost, S.: Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In: *VLDB*, pp. 275–286 (2002)
35. Lee, D.: Query Relaxation for XML Model. Ph.D. thesis. University of California (2002)
36. Mamoulis, N., Papadias, D.: Multiway Spatial Joins. *ACM Trans. Database Syst.* 26(4), 424–475 (2001)
37. Mishra, C., Koudas, N.: Interactive Query Refinement. In: *EDBT*, pp. 862–873 (2009)
38. Navratil, G., Franz, M., Pontikakis, E.: Measurement-Based GIS Revisited. In: 7th AG-ILE Conference on Geographic Information Science, pp. 771–775 (2004)
39. Papadias, D., Arkoumanis, D.: Approximate Processing of Multiway Spatial Joins in Very Large Databases. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) *EDBT 2002*. LNCS, vol. 2287, pp. 179–196. Springer, Heidelberg (2002)
40. Papadias, D., Kalnis, P., Zhang, J., Tao, Y.: Efficient OLAP Operations in Spatial Data Warehouses. In: Jensen, C.S., Schneider, M., Seeger, B., Tsotras, V.J. (eds.) *SSTD 2001*. LNCS, vol. 2121, pp. 443–459. Springer, Heidelberg (2001)
41. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive Skyline Computation in Database Systems. *ACM Trans. Database Syst.* 30(1), 41–82 (2005)
42. Papadias, D., Theodoridis, Y., Sellis, T.K., Egenhofer, M.J.: Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees. In: *SIGMOD Conference*, pp. 92–103 (1995)
43. Peltzer, J.B., Teredesai, A., Reinard, G.: AQUAGP: Approximate QUery Answers Using Genetic Programming. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) *EuroGP 2006*. LNCS, vol. 3905, pp. 49–60. Springer, Heidelberg (2006)
44. Podestà, P.: Query Processing and Analysis of Multi-resolution Spatial Data in Distributed Architecture. Ph.D. thesis. University of Genoa, Italy (2010)
45. Radwan, M., Alvarez, L., Onchaga, R., Morales, J.: The Changing Role of the Geo-Data Infrastructure: from a Data Delivery Network to a Virtual Enterprise Supporting Complex Services. In: *ISPRS 2004 Photogrammetry and Remote Sensing*, pp. 194–199 (2004)
46. Rigaux, P., Scholl, M., Voisard, A.: *Spatial Databases - with Applications to GIS*. Elsevier (2002)

47. Rote, G.: Computing the Minimum Hausdorff Distance Between Two Point Sets on a Line Under Translation. *Inf. Process. Lett.* 38(3), 123–127 (1991)
48. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest Neighbor Queries. In: *SIGMOD Conference*, pp. 71–79 (1995)
49. Shan, J., Zhang, D., Salzberg, B.: On Spatial-Range Closest-Pair Query. In: *SSTD*, pp. 252–269 (2003)
50. Sharifzadeh, M., Shahabi, C.: The Spatial Skyline Queries. In: *VLDB*, pp. 751–762 (2006)
51. Sharifzadeh, M., Shahabi, C., Kazemi, L.: Processing Spatial Skyline Queries in both Vector Spaces and Spatial Network Databases. *ACM Trans. Database Syst.* 34(3), 14:1–14:45 (2009)
52. Silva, Y.N., Aref, W.G., Ali, M.H.: The Similarity Join Database Operator. In: *ICDE*, pp. 892–903 (2010)
53. Skiadopoulos, S., Giannoukos, C., Sarkas, N., Vassiliadis, P., Sellis, T.K., Koubarakis, M.: Computing and Managing Cardinal Direction Relations. *IEEE Trans. Knowl. Data Eng.* 17(12), 1610–1623 (2005)
54. Son, W., Lee, M.-W., Ahn, H.-K., Hwang, S.-w.: Spatial Skyline Queries: An Efficient Geometric Algorithm. In: Mamoulis, N., Seidl, T., Pedersen, T.B., Torp, K., Assent, I. (eds.) *SSTD 2009*. LNCS, vol. 5644, pp. 247–264. Springer, Heidelberg (2009)
55. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient Progressive Skyline Computation. In: *VLDB*, pp. 301–310 (2001)
56. Tian, Y., Lee, K.C.K., Lee, W.C.: Finding Skyline Paths in Road Networks. In: *GIS*, pp. 444–447 (2009)
57. Xia, T., Zhang, D., Kanoulas, E., Du, Y.: On Computing Top-t Most Influential Spatial Sites. In: *VLDB*, pp. 946–957 (2005)
58. Yao, B., Li, F., Hadjieleftheriou, M., Hou, K.: Approximate String Search in Spatial Databases. In: *ICDE*, pp. 545–556 (2010)
59. Yiu, M.L., Dai, X., Mamoulis, N., Vaitis, M.: Top-k Spatial Preference Queries. In: *ICDE*, pp. 1076–1085 (2007)
60. Zhang, J., Mamoulis, N., Papadias, D., Tao, Y.: All-Nearest-Neighbors Queries in Spatial Databases. In: *SSDBM*, pp. 297–306 (2004)
61. Zhu, M., Papadias, D., Zhang, J., Lee, D.L.: Top-k Spatial Joins. *IEEE Trans. on Knowl. and Data Eng.* 17(4), 567–579 (2005)
62. Zimbrão, G., de Souza, J.M.: A Raster Approximation for Processing of Spatial Joins. In: *VLDB*, pp. 558–569 (1998)

Chapter 6

Approximate XML Query Processing

Giovanna Guerrini

Abstract. The standard XML query languages, XPath and XQuery, are built on the assumption of a regular structure with well-defined parent/child relationships between nodes and exact conditions on nodes. Full text extensions to both languages allow Information Retrieval (IR) style queries over text-rich documents. Important applications exist for which the purely textual information is not predominant and documents exhibit a structure, that is however not relatively regular. Thus, approaches to relax both content and structure conditions in queries on XML document collections and to rank results according to some measure to assess similarity have been proposed, as well as processing approaches to efficiently evaluate them. In the chapter, the various dimensions of query relaxation and alternative approaches to approximate processing will be discussed.

6.1 Introduction

Querying XML data on the Web is characterized by data heterogeneity and limited data knowledge. Data heterogeneity means that data referring to the same real-world entity may be contained in distinct data collections and represented in different ways. Heterogeneity in XML data arises from different value representation formats or structures, which may be due to the different provenance of data on processing. Specifically, heterogeneity may appear in the collection at different levels: (i) different tags may be employed in different collections to label the same information; (ii) the hierarchical structure of documents in different sources may be slightly different; (iii) different strings may be employed at content level to represent the same information. Limited data knowledge reflects the fact that the user is not always able to specify the query in a complete and exact way since she may not know all the characteristics of data to be retrieved, even if data come from just

Giovanna Guerrini
Università di Genova, Italy
e-mail: giovanna.guerrini@unige.it

one single source. On these premises, the user cannot claim to get only “precise” answers, that exactly satisfy the search condition expressed by the query, since it is really difficult to exactly characterize what she is looking for and it is quite common that data she is interested in are represented in different ways and thus exhibit different structures.

In such contexts, queries on XML data, expressed through standard XML query languages, XPath and XQuery, that are built on the assumption of a regular structure with well-defined parent/child relationships between nodes, may lead to the empty or few answers problem [2], when the query is too selective or data are quite heterogeneous. In this case, it would be relevant for the user to stretch the query in order to get a set of approximate items as result, ranked according to their relevance to the original query.

An alternative to structure-based database (DB) style XML query languages is represented by Information Retrieval (IR) style queries over text-rich documents. IR-only queries do not pose any structural condition and specify content condition as a set of keywords. The candidate answers to such queries are from the set of lowest common ancestors of tree nodes that correspond to the specified keywords. To support this keyword-based style of querying yet taking advantage of the structure of XML documents, *full text* extensions to XML query languages have been proposed [46]. A *contains* function allows to look for document elements containing approximate matches to keywords. These approaches, however, offer limited support for heterogeneity at the structural level. Thus, from one side we have data-oriented techniques that handle rich XML structures, but do not tolerate a high level of variation in them; on the other side, document-oriented techniques are very good in processing complex textual content, but are limited in their support for complex structural information.

The adoption of XML as a language for the representation and exchange of information by diverse communities – Bioinformatics, cultural heritage, ontologies, GIS and many others – has motivated the appearance of important applications for which none of these querying approaches is effective, since there is neither a relatively regular structure that can be exploited, nor the purely textual information is predominant. For these reasons, several approaches to relax in some way content and structure conditions in queries on XML document collections and approximate processing techniques that are tolerant to significant variability also in document structures have been proposed. In most cases, there will be no exact answer to a given query, but a set of approximate results ranked according to a scoring function, and to avoid the too many answers problem queries are often thresholded or top- k , thus returning only the results whose score is above a certain threshold or among the k highest ranked answers (as discussed in Chapter 2 and in Chapter 3). This requires the ability to rank results according to some measure to assess similarity, and to efficiently process such approximate queries, with the ability of early pruning (partial) answers that would not lead to an answer in the final result set.

Goal of the chapter is to provide an overview of such approaches that allow relaxation of both content and structural conditions. Specifically, we try to highlight

the most relevant issues and discuss in a slightly greater detail some representative approaches. Specific dimensions that will be discussed and along which the approaches will be compared are:

- the type of queries they support,
- the degree of heterogeneity they allow in documents provided as answers to queries, i.e., the relaxation allowed on query conditions,
- the similarity measures proposed to rank results,
- the processing approach proposed for the queries.

We have selected four different approaches: TopX [43], Twig-Path Scoring [4], TASM [7], ArHeX [37]. All of the selected approaches support variations of *twig queries*, that is, tree patterns in which nodes represent the term the user is interested in -content part of the query- and edges represent the structural relationships that the user wants to hold between the terms -structural part of the query. The selected approaches face approximate XML queries in quite different ways: they account for different degrees of heterogeneity, rely on different ranking approaches to assess similarity, and on different processing algorithms.

TopX [43] is a top- k retrieval engine for XML data with XPath Full-Text functionalities, which efficiently supports vague search on both content (also on a semantic base, i.e., through the use of Thesauri and ontologies) and structure-oriented query conditions. The system is however mainly concerned with optimization for content search, while for the assessment of structural similarity they basically count the number of navigational query conditions that are satisfied by a result candidate in their scoring.

Twig-Path Scoring [4] accounts for both vocabulary and structural heterogeneity and consists of scoring methods that are inspired by *tf · idf* and rank query answers on both structure and content. Specifically, twig scoring accounts for all structural and content conditions in the query. Path scoring, by contrast, is an approximation of twig scoring that loosens the correlations between query nodes when computing scores. The key idea in path scoring is to decompose the twig query into paths, independently compute the score of each path, and then combine these scores. An adaptive query processing algorithm to evaluate such approximate matches, in which different plans are permitted for different partial matches, taking the top- k nature of the problem into account is proposed in [32].

TASM [7] addresses the problem of top- k approximate subtree matching, that is, of ranking the k best approximate matches of a *small query* tree (e.g., a DBLP article with 15 nodes) in a *large document* tree (e.g., the whole DBLP with 26M nodes). The k subtrees of the document tree (consisting of nodes of the documents with their descendants) that are closest to the query tree according to canonical tree edit distance are returned. Given the hypothesis, the most crucial issue is to be able to have a runtime cost linear in the size of the document and a space complexity that does not depend on the document size.

ArHeX [37] is an approach for identifying the portions of documents that are similar to a given *structural pattern*, which is a tree whose labels are those that should be

retrieved, preferably with the relationship imposed by the hierarchical structure, in the collection of documents. In the first phase, tags occurring in the pattern are employed for identifying the portions of the documents in which the nodes of the pattern appear. Exploiting the ancestor/descendant relationships existing among nodes in the target, subtrees (named *fragments*) are extracted having common/similar tags to those in the pattern, but eventually presenting different structures. The structural similarity between the pattern and the fragments is evaluated as a second step, for ranking the identified fragments and producing the result.

Since there are many issues and perspectives involved in approximate XML querying, this chapter is by no means exhaustive. First of all, approximate matching for XML data has been considered also in the context of data-to-data matches [23, 9, 8], that are relevant in the context of data integration. Since the focus of the book is on query processing, we do not consider those approaches further in the chapter and rather we focus on query-to-data matches. Moreover, work on approximated ranked XML retrieval as opposed to exact conditions with Boolean answers on XML data started in the early 2000's, as IR-style of retrieval as opposed to DB-style structural querying, and corresponding to the two different views of document-centric rather than data-centric XML documents. As surveyed in [28, 29], approaches in XML search range from keyword only queries, to tag and keyword queries, to path and keyword queries, to XQuery and keyword queries (XQuery Full Text) [46]. The need for integrating such two distinct retrieval modalities emerged soon [13, 6], not only in the XML context. Given the focus of this book, in this chapter we discuss querying rather than retrieval approaches, that is, approaches that takes the structure of the XML document to be queried deeply into account in processing the query, and does not regard it merely as a template to insert keywords in. Specifically, we are interested in approaches that allow to weaken structural constraints and rank structural (partial) conformance, rather than applying them as a filter that, if present, has to be met exactly. One of the approach we survey, namely ArHex, as an extreme case, disregard content at all and focus on document structure only.

The remainder of the chapter is structured as follows. The following subsections discuss the main issues we have identified above (i.e., type of queries, approximation extents, ranking, and processing approaches) with specific reference to the proposals we have selected to focus on. The chapter is concluded by a discussion on some criteria and approaches that can be exploited to choose the best suited way to approximately querying an XML collection at hand, given the degree of heterogeneity (in terms of structure as well as of content) exhibited by the collection.

6.2 Twig Queries

In this section we introduce the tree representation adopted for XML documents and define the notion of twig queries, then we briefly review some of the basic techniques for twig query (exact) processing, and we discuss the types of (twig) queries supported by the selected approximate approaches.

```

<books>
  <book>
    <collection> <title> XML </title> </collection>
    <editor> <name> MK </name> </editor>
  </book>
  <book>
    <editor>
      <name> SA </name>
      <company>
        <address country = "USA"> <city> New York </city>
        </address>
      </company>
    </editor>
  </book>
  <book>
    <author> <name> SA </name> </author>
    <author>
      <name> MK </name>
      <org> <address/ country = "USA"> </org>
    </author>
  </book>
</books>

```

Fig. 6.1 XML document.

6.2.1 XML Documents

An XML document, as shown in Figure 6.1, simply consists of a sequence of nested tagged elements. An element contains a portion of the document delimited by a start tag (e.g., `<author>`), at the beginning, and an end tag (e.g., `</author>`), at the end. Empty elements of the form `<tagname/>` (e.g., `<address/>`) are also possible. The outermost element containing all the elements of the document, element `books` in Figure 6.1, is referred to as document element. Each element can be characterized by one or more attributes, that are name-value pairs appearing just after the element name in the start/empty tag (e.g., `country="USA"`), and by a textual content, that is the portion of text appearing between the start tag and the end tag (e.g., "New York"). XML documents are often represented as ordered labelled trees, where the distinction between subelements and attributes is disregarded and attributes are modelled as subelements with a single textual content. We will rely on this representation in the remainder of the chapter, and we will only talk of elements assuming that attributes are represented as subelements. Most querying approaches, moreover, neglect links between document elements, that would make the document a graph rather than a tree. We will not consider links in what follows either. In this way, internal tree nodes are labelled by element tags while leaves are labelled by data content. Figure 6.2 contains the tree representation of the XML document in Figure 6.1

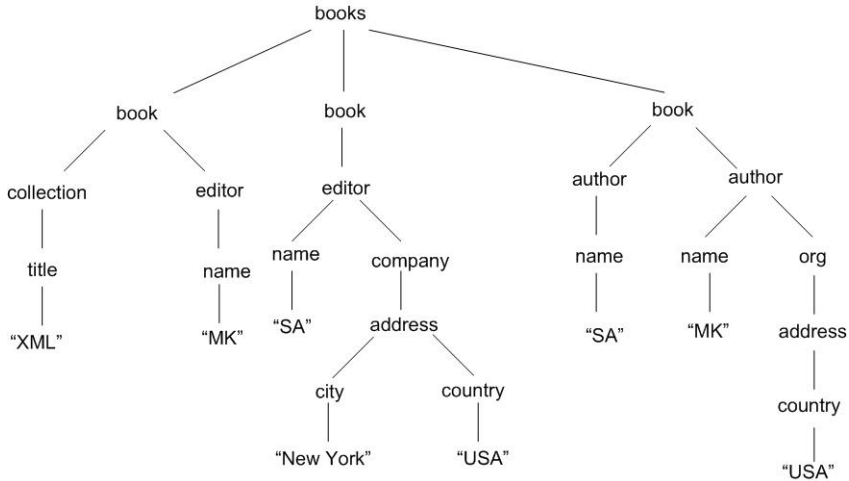


Fig. 6.2 Tree representation of the XML document of Figure 6.1

6.2.2 Twig Queries: Definition

Several languages have been proposed for querying XML documents. Specifically, XPath [44] and XQuery [45] have been accepted as standards by the W3C. Though a full text extensions of XQuery has been proposed, XQuery Full Text [46], most approaches focused on a rather restricted set of queries known as *twig* or *tree pattern* queries and we focus on this class of queries in this chapter.

A twig query is a rooted labelled tree with two types of edges: /, child edge, and //, descendant edge. A child edge represents a downward edge in the XML document tree whereas a descendant edge represents a downward path in such a tree. Internal node labels are element tag names whereas leaf node labels can be tag names as well as predicates or strings, in which case they are interpreted as the predicate “value equals to the string label”. A twig pattern corresponds to a core of XPath [44] with only child and descendant axes.

A distinguished node can be explicitly marked as the target of the query [5]. If not explicitly marked, the root of the query tree is the target node. The various branches in the tree represent multiple predicates on the corresponding subtree.

Example 1. Figure 6.3 (a) shows an example of twig query, asking for books elements with a *collection* child containing “XML” in its full content (i.e., content of one of its arbitrarily nested subelements) and an *editor* child whose name is “SA” and having a descendant *address* subelement containing “USA” in its full content. The XPath expression corresponding to this twig query is `//book[collection//*[.='XML']][editor/name='SA'][editor//address//*[.='USA']]`.

By contrast, the twig query in Figure 6.3 (b) asks for books elements with a *collection* child with “XML” value and an *editor* child whose name is

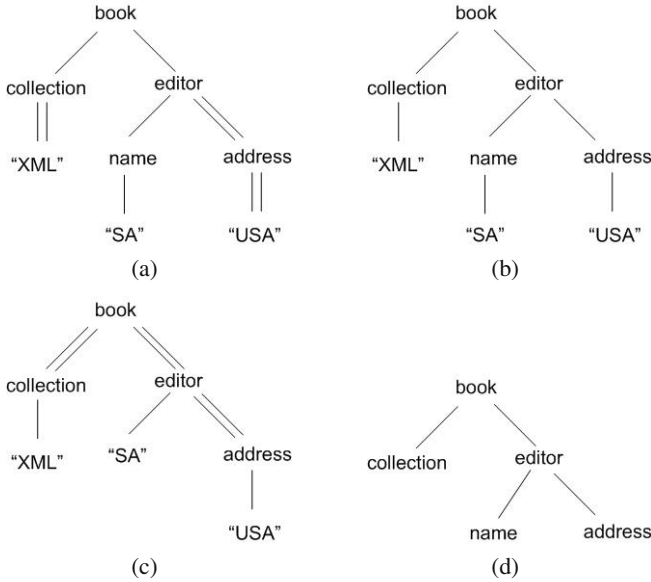


Fig. 6.3 Twig query (a), twig query with only child edges (b), twig queries with only descendant edges (c), structural pattern (d).

“SA” and whose address is “USA” (with name and address direct subelements of editor). The XPath expression corresponding to this second query is: `//book [collection='XML'][editor/name='SA'][editor/address='USA']`.

If, by contrast, in this second example, we were not interested in imposing any constraints on the editor address but would rather like to be returned this information than the `book` element, the `address` node in the twig would have no child and would be marked as distinguished element, and the corresponding XPath expression would be: `//book [collection='XML']/editor [name='SA']/address`.

The term *match* denotes the assignments of query nodes to document nodes that satisfy the constraints imposed by the query and the term *answer* denotes document nodes for which there is a match that maps the target node (i.e., the root in most common approaches) of the query to such a node. Note that no match for the query in Figure 6.3 (a) exists on the document of Figure 6.2. By contrast, if we consider the query without the subtree rooted at `collection` there is a match with the central `book` child of the target document, which is returned as an answer.

6.2.3 (Exact) Twig Query Processing

Twig pattern matching is essential in evaluating XPath/XQuery queries, and thus several different approaches for processing (exact) twig queries have been proposed

(see [19] for a survey). Generally, pre-built indexes on XML data can facilitate twig query processing by locating target data avoiding exhaustive scans of all the data. Indexes types include value indexes (as classical B+-tree indexes), which index data values in the documents, and structural indexes, which index the structure of the documents. Most relevant structural indexes are instances of two broad classes: numbering schemes and index graph schemes, also called structural summaries. Index graph schemes are mainly used for path selection and numbering schemes for path joining. Numbering schemes allow to determine structural relationships among nodes without tree navigation. Among the best known numbering schemes we mention the Pre- and Post-order encodings of elements within document trees [20] and the Dewey encoding [41] which is easier to maintain under dynamic updates. The idea of structural summaries, by contrast, is to summarize an XML data tree into a smaller structure that can be queried more efficiently, by summarizing path information. An early example of such approaches is the Strong Data Guide [18]. Structural indexes have been extended so to support answering twig queries in, e.g., the *F&B*-index [1] that partitions data nodes into equivalence classes based on forward and backward bi-similarity among nodes.

For what concerns the processing itself, approaches can be categorized into two classes: the relational approach, by which data are loaded into relational databases and twig queries are transformed into SQL queries over relational data, and the native approach, by which XML data are stored on disk in the form of inverted lists, sequences, or trees and native algorithms are developed to further improve XML twig query performance. Among the native approaches, that include various forms of structural joins, multi-predicate merge joins, let us just mention the staircase join based on PrePost encoding [20], and holistic twig joins [11]. The latter, also known as path stack algorithm, is probably the most efficient method for twig queries using a combination of sequential scans over index lists stored on disk and linked stacks in memory.

6.2.4 Twig Queries as a Basis for Approximate Querying

All the approaches we discuss in this chapter rely on (variations of) twig queries as query language. Table 6.1 summarizes the variations of twig queries supported by the four approximate querying approaches we discuss more deeply in this chapter. Twig-Path Scoring [4, 32] considers twig queries with distinguished root node, string labelled nodes, and both child and descendant edges. In TASM [7] queries are tree patterns with distinguished root node, string labelled nodes, with tags appearing only at internal nodes (and content at leaves) and only child edges (as the one in Figure 6.3 (b)).

In TopX [43] queries can be evaluated in two distinct modalities, that can be configured orthogonally to the query formulation, so to return two different granularities as result. In *document mode*, TopX returns whole documents as answer to a query

Table 6.1 Types of tree pattern queries.

	Twig-Path Scoring	TopX	TASM	ArHeX
query answer	document subtree ¹	document or document subtree	document subtree ¹	region
edges	child and descendant	descendant	child	child
node labels	strings	strings and predicates	strings (no tag names at leaves)	strings (tag names)

¹ Rooted at a node matched to the query root.

whereas in *element mode* it returns target elements, that may be from the same as well as from different documents. Only descendant edges are considered, whereas multiple content-labeled leaves can be associated as children with a node (as in Figure 6.3 (c)). This also corresponds to the fact that the implicit predicate associated with such strings is not equality but rather the IR-style `about` operator (analogous to `contains` of XPath Full-Text). In addition, a \sim operator can be employed in labels to denote user-requested semantic expansion of the argument term/tag.

In ArHex [37], since the focus is on structure, only tag labelled nodes can appear in queries. Thus, the considered queries (named structural patterns or twigs) are tree patterns with only child edges and string labelled nodes, where labels correspond to element tags only (no conditions are imposed on data content). An example is given in Figure 6.3 (d). As a consequence, the answer is not a complete subtree (that is, reaching the leaves). Rather, it is a (minimal) subtree, referred to as *fragment*, only including the relevant nodes, i.e., nodes matching some query node. Such nodes can be anywhere in the hierarchical structure of the document and several edges in the original document may be collapsed in a single edge. Actually, since a larger document portion, obtained by merging different fragments, may exhibit higher similarity to the query than the fragments it originates from considered separately, the answer to the query is a document *region*. The root of the returned subtree is a node not necessarily matching the query root.

When twig queries are evaluated approximately, there is the need to limit the number of results. Either the allowed query relaxation are explicit in the query, and all the answers to the relaxed queries are returned, or queries are formulated as *threshold* or as *top-k* queries. In all the cases, answers come with a rank, or score, measuring how closely the answer meets the original query. Usually, the rank or score is a number between 0 and 1, with the intuition that a precise answer is scored 1, while 0 denotes no (even approximate) match. Results are provided in descending rank order. In threshold queries a threshold ε is specified, and all the answers whose score is above ε are returned. In *top-k* queries, a value k is specified and the k highly scored answers are returned. In both cases, it is crucial to early prune subtrees that will not result in a score above the threshold/in the highest k scores.

6.3 Various Extents of Approximation

In this section we present different ways in which the exact match semantics of twig queries can be relaxed or approximated, distinguishing between vocabulary and hierarchical structure issues, and among explicitly requested, limited or arbitrary approximation. The considered approaches are then situated in the resulting approximation framework in Table 6.2.

6.3.1 Vocabulary

Vocabulary approximation refers to the type of approximate match introduced at string level, which may be different for element tags (strings appearing as labels of internal nodes) and document content (strings appearing as labels of leaf nodes). Different types of approximate match can be considered instead of equality, for both types of nodes. Specifically, *stemming* [31] reduces inflected (or sometimes derived) words to their stem, base or root form. For instance, *stemmer*, *stemming*, *stemmed* are examples of words based on the root *stem*. The stem needs not be identical to the morphological root of the word; it is usually sufficient that related words map to the same stem, even if this stem is not in itself a valid root. Words with the same stem are considered as matching, even if they are not equal as strings. Thesauri could be exploited as well to assess string similarity across different languages.

Another alternative is to consider two strings as matching, even if they are not equal, provided they can be recognized as *synonyms*, that is, different as terms but with the same meaning, according to some Thesaurus, such as WordNet [17]. Note that this requires to be able to associate an intended meaning with a string, and in particular to cope with *polysemy*, that is, a term that may have different meanings depending on the context that it is used in. Some simple techniques for Word Sense Disambiguation (WSD) [25] relying on the context the string appears in have to be used unless we want to cope with multiple possible meanings, and thus multiple possible *synsets* (i.e., sets of synonyms), associated with the same term.

In addition to the above two *semantic* similarity notions for words, a more *syntactic* approximation can be allowed on strings, relying on any notion of string similarity, such as string edit distance or q-gram based distances [34] to account for small syntactic differences among strings. Though syntactic approximation seems much less interesting than semantic one, we have to consider that as tag often a combination of lexemes (e.g., `ProductList`, `SigmodRecord`, `Act_number`), a shortcut (e.g., `coverpg`, `cc`), a single letter word (e.g., `P` for paragraph, `V` for verse), a preposition or a verb (e.g., `from`, `to`, `related`) rather than a noun is used.

Example 2. Referring to the document of Figures 6.1 and 6.2, queries containing the following tags/terms would fail to produce an exact match, but they will produce a match if element tags/content were approximately matched. As examples of stemming: `authors` vs `author`, `collect` vs `collection`; as an example of synonyms: `monograph` vs `book`; as an example of cross-language similarity: `libro` vs `book`; as an example of syntactic similarity `U . S . A .` vs `USA`.

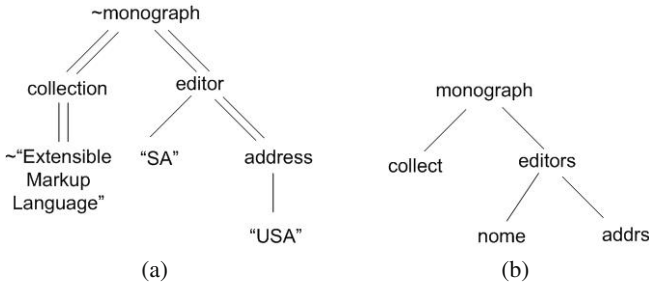


Fig. 6.4 Vocabulary variations: explicitly requested (a) and implicitly applied (b).

The approximation on node labels to account for vocabulary heterogeneity may be implicitly applied or explicitly requested by the user in formulating the query. Specifically, in TopX, stemming and synonyms according to the WordNet Thesaurus are accepted as matches, both for tags and for data content, if the \sim operator is explicitly used in query formulation. Moreover, the same idea is automatically applied by relaxing the equality predicates at nodes during incremental query expansion, for queries for which not enough results can be found. To choose among the possible meanings of the word which is the right one, they rely on word statistics for some local context of both the terms that appear in a part of a document or a query and the candidate meanings that are extracted from the ontology graph containing the concepts. The idea of accepting and evaluating semantic similarity between terms in XML queries through a similarity enhanced ontology has been proposed also in TOSS [24]. The ontology captures inter-term lexical relationships and can use any measure of semantic similarity among terms.

Vocabulary heterogeneity, both at a syntactic and at a semantic level is considered in ArHeX, obviously only for tags, since queries in ArHeX are structural queries with no data content. By contrast, in Twig-Path Scoring [4] they explicitly state that approximated matches at nodes are orthogonal to and beyond the scope of their work. The issue of node label similarity is not considered by TASM either, in that any node relabeling is allowed and the same (unit) cost is applied to any node relabeling. Thus, neither semantic nor syntactic similarity is accounted for.

Example 3. Referring to the queries in Figure 6.4, TopX would return for the query in Figure 6.4 (a) the same answers as for the query in Figure 6.3 (c), and ArHeX would return for the structural query in Figure 6.4 (b) the same answers as for the structural query in Figure 6.3 (d). Any modifications to the labels of nodes of the twigs in Figure 6.3 (a) and Figure 6.3 (b) would result in a node mismatch in Twig-Path Scoring and TASM, irrespective on how different the new label is from the original one (syntactically and/or semantically).

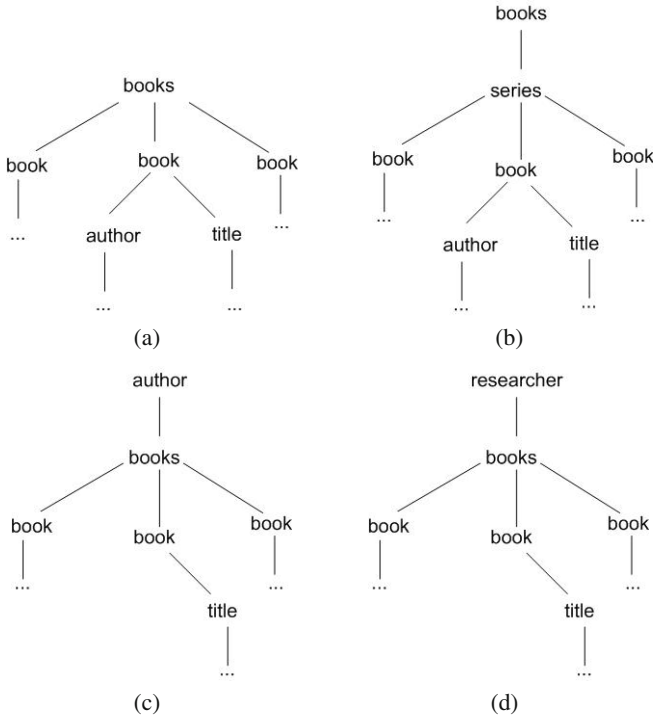


Fig. 6.5 Structural variations.

6.3.2 Hierarchical Structure

Structural approximation may be allowed as well, allowing for arbitrary variations in the structure or through controlled relaxation of structural constraints. Consider for instance the queries in Figure 6.5 with respect to the actual document structure illustrated by the tree in Figure 6.5 (a). In Figure 6.5 (b), an intermediate node *series* appears between nodes *books* and *book* that are thus bound by an ancestor-descendant rather than by a parent-child relationship. In Figure 6.5 (c), the ancestor-descendant relationship between *author* and *book* (and also *books*) is reversed, corresponding to a different organization in which *books* are represented grouped by *authors*. In Figure 6.5 (d), in addition to the above structural variation, there is also a vocabulary variation since *researcher* is used instead of *author*.

In Twig-Path Scoring [4, 32] the same structural relaxations of [5] are considered:

- edge generalization:* replacing a child edge / with a descendant edge //,
- leaf deletion:* removing a leaf node from the twig,
- subtree promotion:* moving a subtree from its parent node to its grand-parent.

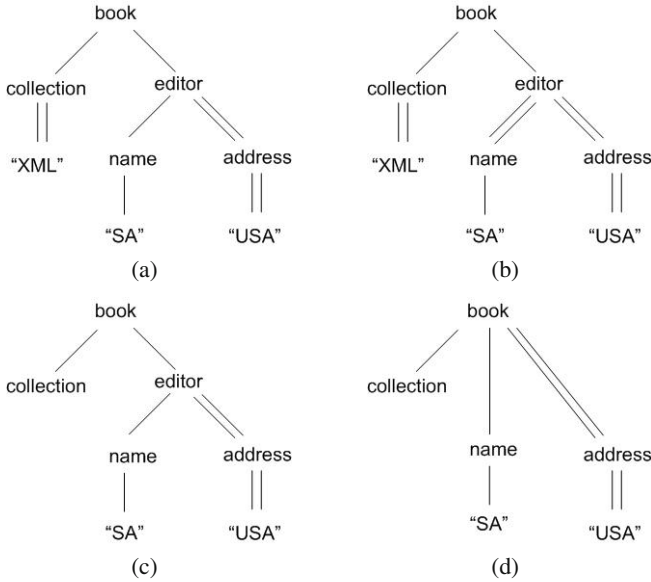


Fig. 6.6 Structural query relaxation.

These relaxations generalized the ones previously proposed in [3] and [39]. For instance, in [3] relax node, delete node, relax edge, and promote node are considered as possible relaxations whereas in [39] node insertions and node deletions are considered.

Example 4. Referring to the twig query in Figure 6.6 (a), an example of edge generalization is given in Figure 6.6 (b) in which the edge between `editor` and `name` has been relaxed. An example of leaf deletion is given in Figure 6.6 (c) in which the condition on the `collection` content has been removed. An example of subtree promotion is given in Figure 6.6 (d), in which the `editor` node has been deleted and its children promoted as children of the root.

Structural approximations allowed in the answer with respect to the structure specified in query pattern are not restricted in the other approaches. However, in TopX, any path condition is evaluated in its entirety as matched or not matched, and partial matches of nodes in the path are disregarded. By contrast, in TASM partial structural matches are considered, since the match is evaluated at a node basis (with a unit cost model that assigns the same cost to node deletion, insertion, and relabeling). Matches (referred to node alignments) are however required to be consistent with the ancestor-descendant and sibling order relationships. Partial structural matches are considered in ArHeX as well, which also evaluates matches on a node basis, but do not impose any restrictions to the matches among nodes (provided their tags are equal or similar according to the devised criterion).

Table 6.2 Extent of approximation.

	Twig-Path Scoring	TopX	TASM	ArHeX
vocabulary	–	semantic on tags and terms	any relabeling has unit cost	syntactic and semantic on tags
structural	edge generalization leaf deletion subtree promotion	patch matches are considered only if exact	any edit has unit cost ancestor-descendant relationships and sibling order are preserved	arbitrary

Example 5. Referring to the structural variations of Figure 6.5, all the approaches would consider document portions like that in Figure 6.5 (b) when looking for a structure like the one in Figure 6.5 (a). All the approaches but TopX would consider document portions like that in Figure 6.5 (a) when looking for a structure like the one in Figure 6.5 (b). In TopX, by contrast, the paths `books//series//book//author` and `books//series//book//title` would be evaluated as not matched since the node `series` is missing. Only ArHeX, finally, would consider document portions like those in Figure 6.5 (c) and Figure 6.5 (d) when looking for a structure like the one in Figure 6.5 (a).

6.4 Ranking

A consequence of allowing for approximate answers to a query is the need for a ranking or scoring method, able to assess similarity between the answer and the original request. An obvious requirement is that an exact answer (exact match) will be scored higher than an approximate one (partial match). A variety of IR-style scoring functions has been proposed and adopted for XML retrieval, ranging from the classic vector space model with its *tf · idf* family of scoring approaches [35, 10], typically using Cosine measure for score aggregations, over to the theoretically more sound probabilistic scoring models, such as Okapi BM25 [27], among the most widely used ranking approaches in current IR benchmark settings such as TREC or INEX. Similarity measures for XML documents have been proposed in very diverse contexts, ranging from data integration to document clustering and change detection. A complete discussion of this topic is beyond the scope of this chapter. Extensive surveys can be found in [21, 22, 42]. In this section we present the ranking methods employed by the approaches we focus on, that are however quite different and quite representative of alternative approaches that can be employed.

In devising an adequate assessment of similarity there are different aspects to be traded-off: besides accuracy, efficiency of the ranking function is important as well. This is the reason why some early approaches assign precomputed weights at nodes and then combine them according to the tree structure.

In what follows we discuss the ranking methods of the four approaches dealt with in greater detail in this chapter. TASM relies on a purely tree edit distance ranking model, whereas ArHeX supports different ranking models, but basically relies on the idea to evaluate the goodness of an answer on the basis of the matching portions in the two trees only, TopX and Twig-Path Scoring, by contrast, take into account in the ranking more IR-style notions such as measures of specificity of tags/terms in the whole document/document set and not only the matched document portions (and their structural relationships). While TopX emphasis is on content scoring, structure scoring is preeminently addressed by Twig-Path Scoring. Table 6.3 summarizes these main differences.

6.4.1 Tree Edit Distance

Tree edit distance [40, 48] is a well-known measure to quantify the distance among arbitrary ordered labelled trees, with applications ranging from the comparison of RNA secondary structures to syntax theory, where to compare two sentential forms the distance between their parse trees can be computed. The editing operations available in the tree editing problem are changing (i.e., relabeling), deleting, and inserting a node. To each of these operations a cost is assigned, that can depend on the labels of the involved nodes. The distance between trees T_1 and T_2 is defined to be the cost of the minimum cost sequence of such operations transforming T_1 into T_2 . Three kinds of operations for ordered labelled trees are considered. Relabeling a node n means changing the label on n . Deleting a node n means making the children of n become the children of the parent of n and then removing n . Inserting n as the child of m will make n the parent of a consecutive subsequence of the current children of m .

Edit operations can be defined in terms of edit mappings, such that: each node is mapped, the mapping is one-to-one, ancestor-descendant relationships are preserved, sibling order is preserved. The mapping is represented by a set of node alignments. Let ε be a unique node not in the set of nodes, denoting the null node. An edit operation is represented as a *node alignment* $a \rightarrow b$, where a is either ε or a node in T_1 and b is either ε or a node in T_2 . An operation of the form $\varepsilon \rightarrow b$ is an insertion, an operation of the form $a \rightarrow \varepsilon$ is a deletion. Finally, an operation of the form $a \rightarrow b$, with $a, b \neq \varepsilon$, is a relabeling. Assuming each node n be assigned a cost $c(n) \geq 1$ the cost γ of a node alignment $a \rightarrow b$ can be defined as (i) $\gamma(\varepsilon \rightarrow b) = c(b)$ (insertion); (ii) $\gamma(a \rightarrow \varepsilon) = c(a)$ (deletion); (iii) $\gamma(a \rightarrow b) = c(a) + c(b)/2$ if node a and b labels are different (relabeling); (iii) $\gamma(a \rightarrow b) = 0$ if node a and b labels are equal (no change).

Function γ is extended to an edit mapping, consisting of a set of node alignments $M = s_1, \dots, s_k$ by letting: $\gamma(M) = \sum_{i=1}^k \gamma(s_i)$. The edit distance between the two trees T_1 and T_2 is defined as the minimum cost edit mapping that transforms T_1 to T_2 , that is: $D(T_1, T_2) = \min_M \{\gamma(M) | M \text{ is an edit mapping from } T_1 \text{ to } T_2\}$.

For instance, assuming as in [7] unit costs for all the edit operations, the twig query in Figure 6.3 (b) has distance 4 from the leftmost `book` node of Figure 6.2: one node deletion (`title`), a node relabeling ("`MK`" to "`SA`"), and two node insertions (`address` and "`USA`"). By contrast, it has distance 6 from the central `book` node of Figure 6.2: two node insertions (`collection` and "`XML`") and four node deletions (`company`, `city`, "`New York`", `country`). Note that in this way the approach is estimating the distance rather than the similarity and thus, in this case, the best answers are those having lowest distance from the query (and exact answers have distance 0).

The best known and reference approach to compute edit distance for ordered trees is by Zhang and Shasha [48]. Their dynamic programming algorithm recursively decomposes the input trees into smaller units and computes the tree distance bottom up. The decomposition may produce forests. The algorithm computes the distance between all pairs of subtree prefixes of two trees.

6.4.2 *An Alternative Match Based Similarity*

ArHeX employs as well a notion of similarity that relies on the evaluation of a mapping between nodes. Since the focus is on heterogeneous semi-structured data, however, the hierarchical organization of the query and the answer are not taken into account in the definition of the mapping, the only requirement is that the element labels are similar. Note thus that an important difference is that while the TASM approach poses some constraints on the allowed node alignments (preservation of ancestor-descendant relationships and of sibling order) on which tree edit distance is then evaluated (according to unit cost model), in ArHeX matching only relies on the identification of nodes with the same or similar tags, and the degree of structure conformance is only taken into account in ranking the matches.

Several mappings can be established between a query and an answer region. The best one will be selected by means of a similarity measure that evaluates the degree of similarity between the two structures relying on the degree of similarity of their matching nodes. The evaluation of a mapping is defined by the sum of the similarities of nodes in the mapping, that can be computed according to different similarity measures, as discussed below, normalized by the number of nodes in the query. The similarity between a query and an answer is then defined as the maximal evaluation among the mappings that can be determined between them (analogously to what we have discussed for tree edit distance).

Three different approaches to evaluate similarity among nodes in the mapping are proposed in [37]. The first one assesses similarity only on the basis of label matches, whereas the other two take the structure into account. In *match-based similarity*, similarity only depends on node labels. Similarity is 1 if labels are identical, whereas a pre-fixed penalty δ is applied if labels are similar. Any other syntactic or semantic evaluation of similarity in the range $[0, 1]$ could be considered. If they are not similar, similarity is 0.

In *level-based similarity*, the match-based similarity is combined with the evaluation of the levels at which the two nodes in the mapping appear in the query and in the answer. Whenever they appear in the same level, their similarity is equal to the similarity computed by the first approach. Otherwise, their similarity linearly decreases as the number of levels of difference increases.

Since two nodes can be in the same level, but not in the same position, a third approach is introduced, called *distance-based similarity*. The similarity is computed by taking the distance of nodes in the mapping with respect to their roots into account. Thus, in this case, the similarity is the highest only when the two nodes are in the same position (i.e., taking into account also order among siblings) in the query and in the answer.

As an example, consider the structural twig of Figure 6.3 (d) matched against the document in Figure 6.2 and the three fragments labelled at a `book` element. In the ranking, the leftmost one is evaluated more similar to the twig than the central one that is, in turn, more similar to the twig than the rightmost one. The first one, indeed, only lacks a level-2 element (i.e., `address`). The second one lacks a level-1 element (i.e., `collection`) and an element (i.e., `address`) appears at a different level than expected. In the third one, besides the lack of a level-1 element (i.e., `collection`) and the level mismatch for `address`, there is also tag similarity rather than equality between `author` and `editor` to be taken into account (or the mismatch of the two nodes if they are not deemed similar).

6.4.3 Structure and Content $tf \cdot idf$ Scoring

The scoring function proposed in [4] (Twig-Path Scoring) is based on the principle that best answers are matches to the least relaxed query in the graph of query relaxation. The scoring function is based on the $tf \cdot idf$ (term frequency and inverse document frequency) weight proposed in IR [35] to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. In [4], the idf scoring is however modified to guarantee that answers to less approximate queries obtain idf scores at least as high as scores to more approximate ones. The idf score for an answer e is defined as the maximal idf value of all relaxations of the query having e as an answer. This is also the basis for assuring that the score monotonicity requirement is met by the overall score, that is, that more precise answers to the user query are assigned higher scores. Intuitively, the idf measure of a query Q quantifies the extent to which answers to the most relaxed query additionally satisfy Q . Thus, more selective queries are assigned higher idf scores. This is analogous to the IR case: keywords that appear in a document collection less frequently are assigned higher idf scores. Note, however, that the idf measure defined above assigns the same idf score to all exact matches to a query. In general, all answers having their best match with respect to the same relaxed query are given the same idf score. The idf scores are then used to rank relaxed matches based on how closely they match the relaxed query. To distinguish between matches

of the same relaxed query the analogue of the term frequency tf measure is used. Intuitively, the tf score of an answer quantifies the number of distinct ways in which an answer matches a query. This is again analogous to the IR case, where the term frequency increases with the number of occurrences of a keyword in a document. The final scoring function for twig queries is based on combining the idf and tf scores. A lexicographical scoring (idf,tf) is used to satisfy the score monotonicity requirement.

The twig scoring as specified above requires however to have access to the idf scores associated to all relaxations of the original query to compute the scores of answers. Computing (or even pre-computing whenever possible) these scores can be very expensive. Thus, in order to improve efficiency of the overall query processing, approaches based on decomposing an original twig query to simpler queries are considered and in this way the number of different idf scores needed is reduced. Also, in many cases the scores for such simpler queries are easier to compute. Specifically, two decompositions for a twig query Q are considered:

- *Path Decomposition* the set of all paths in the query leading from the root to any other node; and
- *Binary Decomposition* the set of all queries $root_Q/m$ or $root_Q//m$ for a node m in Q such that they subsume Q .

The score of an answer is computed taking occurrences of all structural and content-related (i.e., keyword) predicates in the query. For example, a match to query of Figure 6.3 (a) would be assigned an inverse document frequency score, idf , based on the fraction of the number of `book` nodes with a `collection` child containing “XML” in its full content and an `editor` child whose name is “SA” and having a descendant subelement `address` containing “USA” in its full content. The match would then be assigned a term frequency score, tf , based on the number of query matches for the specific book answer.

6.4.4 Content Scoring with Structure Filters

Ranking in TopX is based on the following components:

- Content related query conditions are split into combined tag-term pairs. Each matched tag-term pair obtains a pre-computed IR-style relevance score, as discussed below.
- Hierarchical structural conditions are split into single node tests. Each navigational query condition that is not part of a tag-term pair contributes to the aggregated score of a matched subtree in a document by a static score mass c if *all* transitively expanded structural constraints rooted at it can be matched.

The score is computed differently depending on whether the query is evaluated in document or element mode, but it is anyway computed as the maximum score of the matched target element.

As far as the first component, i.e., content score, is concerned, the relevance of a tag-term pair is computed relying on the following measures:

- $ftf(t, e)$ (full-content term frequency) models the relevance of a term t for an element full content, i.e., the frequency of t in all the descending text nodes of element e ;
- N_A (tag frequency) is the number of elements with tag name A in the entire set of documents;
- $ef_A(t)$ (element frequency) models the specificity of t for a particular element with tag name A by capturing how many times t occurs under a tag A across the whole collection having N_A elements with this tag name.

The basic idea is thus to evaluate tag-terms pairs so that the score of an element e with tag A and content condition requiring the occurrence of term t should reflect: (i) the ftf value of the term t , reflecting the occurrence statistics of the term for the element content; (ii) the specificity of the search term, with respect to tag-name specific $ef_A(t)$ and N_A statistics; and (iii) the size and thus the compactness of the subtree rooted at e containing t in its full-content. This leads to the following template:

$$score(e, A = t) = \frac{occurrence \cdot specificity}{size(e)}$$

The TopX engine adapts the empirically very successful Okapi BM25 probabilistic scoring model [27] to a generic XML setting by computing individual relevance models for each element type occurring in the collection. For an `about` operator with multiple keyword conditions that is attached to an element e with tag name A , the aggregated score of e is simply computed as the sum of the elements scores over the individual tag-term conditions.

The structural scoring model essentially counts the number of navigational (i.e., tag-only) query conditions that are satisfied by a result candidate and thus connect

Table 6.3 Ranking

	Twig-Path Scoring	TopX	TASM	ArHeX
base of the scoring	matches for query relaxations in the document	connected document subtree that contains the query target element	node alignment, node and op costs	node match, tag similarity measure
locality ¹	global	global	local	local
binary evaluations	on labels (tags and content)	on paths	on labels (tags and content)	–
preeminent view	structure	content	structure	tags

¹ Local denotes that the evaluation depends only on the query and on the document portion (answer) under evaluation, global that it also depends on the whole document to account for frequencies of labels and/or structural patterns.

the content conditions matched. A navigational condition is fully matched by an element in a document if all the structural constraints, i.e., the element outgoing edges, are satisfied. A structural score c is assigned to any matched navigational condition (element-to-leaf path). Score c is tunable to allow to balance in the most appropriate way structural score and content score. In the scoring model described in [43] the setting of score c is relatively crude and the possibility of making c dependent on the goodness of an approximate structural match is mentioned as a possible direction to explore.

6.5 Approximate Query Processing

The techniques for approximate query processing are very diverse and are deeply influenced by the degree of approximation accounted for. For instance, some early approach [14] applies all allowed transformations to data on querying, but this approach is viable only if the number of possible transformations is quite limited. Other approaches evaluate all the possible relaxations of the original query, but, again, this is possible only if they can be obtained on the basis of the query only. In [39], they take advantage of the presence of a schema for limiting possible relaxations to generate and to evaluate.

An important issue to consider is that approximate queries on XML data are often thresholded or top- k queries. In such cases, the ability to early prune candidate answers that are unlikely to produce highly scored results is crucial for an efficient processing. In what follows we sketch the basic ideas on which query processing relies in the reference approaches discussed in the chapter. XML top- k query evaluation techniques are explicitly discussed in [33] and [26].

6.5.1 Twig-Path Scoring and Whirpool

A typical top- k algorithm is employed in [4] to find the top- k elements based on the score discussed in Section 6.4. Specifically, a DAG maintains pre-computed *idf* scores for all possible relaxed queries that a partial match may satisfy. A matrix representation is employed for queries, their relaxations, and partial matches to quickly determine the relaxed query that is best satisfied by a partial match during top- k query processing and prune irrelevant partial query matches.

Whirpool [32] is a flexible architecture for processing top- k queries on XML documents adaptively. The approach allows partial matches to the same query to follow different execution plans, and takes advantage of the top- k query model to make dynamic choices during query processing. The key features of the processing approach are: (i) a partial match that is highly likely to end up in the top- k set is processed in a prioritized manner, and (ii) a partial match unlikely to be in the top- k set follows the cheapest plan that enables its early pruning.

The adaptive query evaluation relies on *servers*, one for each node in the twig pattern. One of these servers is special in that it generates candidate matches to the root of the query, which initializes the set of partial matches that are adaptively routed through the system. Each of the other servers, for an element e , maintains a priority queue of partial matches (none of which has previously gone through this server). For each partial match at the head of its priority queue, it (i) computes a set of extended (partial or complete) matches, each of which extends the partial match with an e node (if any) that is consistent with the structure of the queries, (ii) computes scores for each of the extended matches, (iii) determines if the extended match influences or is influenced by the top- k set. The system maintains a candidate set of top- k (partial or complete) matches, along with their scores, as the basis for determining if a newly computed partial match: (i) updates the score of an existing match in the set, or (ii) replaces an existing match in the set, or (iii) is pruned, and hence not considered further. Matches that are complete are not processed further, whereas partial matches that are not pruned are sent to the router. The matches generated from each of the servers, and not pruned after comparison with the top- k set, are sent to the router, which maintains a queue based on the maximum possible final scores of the partial matches over its input. For the partial match at the head of its queue, the router determines the next server that needs to process the partial match, and sends the partial match to the queue of that server.

6.5.2 TopX

The TopX engine [43] operates over a combined inverted index for content- and structure-related query conditions by pre-computing and materializing joins over tag-term pairs. This simple pre-computation step makes the query processing more scalable, with an encoding of the index structure that is easily serializable and can directly be stored sequentially on disk just like any inverted index, for example using conventional B+-tree indexes or inverted files.

At query processing time, TopX scans the inverted lists for each tag-term pair in the query in an interleaved manner, thus fetching large element blocks into memory using only sorted access to these lists and then iteratively joining these blocks with element blocks previously seen at different query dimensions for the same document. Using Pre-/Post-order tree encodings [20] for the structure, TopX only needs a few final random accesses for the potential top- k items to resolve their complete structural similarity to a path query. An extended hybrid indexing approach using a combination of Data Guide-like path indexes and Pre-/Post-order-based range indexes (like those discussed in Section 6.2.3) can even fully eliminate the need for these random accesses however at the cost of more disk space.

TopX further introduces extensions for probabilistic candidate pruning, as well as a probabilistic cost-model for adaptively scheduling the sorted and random accesses, that help to significantly accelerate query evaluations in the presence of additional, pre-computed index list statistics such as index list selectivities, score distribution histograms, or parameterized score estimators, and even index list (i.e., keyword)

correlations. For dynamic expansions of tag and term conditions, TopX can incrementally merge the inverted lists for similar conditions obtained from a background Thesaurus such as WordNet.

6.5.3 TASM

TASM [7] focuses on finding the k best approximate matches of a *small* query tree Q in a *large* document tree T . The k subtrees of T (consisting of nodes of T with their descendants) that are closest to Q , according to canonical tree edit distance, are returned. The naive solution to this problem requires $O(m^2n^2)$ time and $O(mn)$ space where m and n are query and document sizes, respectively. This can be improved to $O(m^2n)$ by computing the distance between Q and T with a dynamic programming approach and ranking the subtrees of T visiting the resulting table.

The TASM approach is based on a prefix ring buffer that performs a single scan of the document. The size of the prefix ring buffer is independent of the document size. They rely on a post-order queue that uses the post-order position and the subtree size of a node to represent the document structure. The post-order queue is a sequence of $(label, size)$ pairs of the tree nodes in post-order, where *label* is the node label and *size* is the size of the subtree rooted in the node.

Their top- k algorithm relies on the observation that there is an effective bound on the size of the largest subtrees of a document that can be in the top- k best matches to a query. Pruning large subtrees efficiently and computing tree edit distance on small subtrees only (for which the computation is unavoidable) give rise to an efficient solution to the problem. The pruning algorithm uses a prefix ring buffer to produce the set of all subtrees that are within a given size threshold τ , but are not contained in a different subtree also within the same threshold. This set of *candidate trees* can be computed for a given size threshold τ dequeuing nodes from the post-order queue and appending them to a memory buffer. Once a candidate subtree is found, it is removed from the buffer, and its tree edit distance to the query is computed.

Nodes in the post-order queue are either candidates (i.e., belong to a candidate tree and must be buffered) or non-candidates nodes (root of subtrees too large for the candidate set). A simple pruning approach is to append all incoming nodes to the buffer until a non-candidate node nc is found and then all subtrees rooted in nc children that are smaller than τ are candidate subtrees. An improvement is ring buffer pruning which buffers candidate trees only as long as needed and uses a look-ahead of only $O(\tau)$ nodes. This buffer is moreover enriched with a prefix array which encodes tree prefixes and allows the leftmost valid subtree to be found in constant time.

6.5.4 ArHeX

The main issue in ArHeX is how to efficiently identify fragments, that are portions of the target containing labels similar to those of the pattern, without relying on

strict structural constraints. The proposed approach employs ad-hoc data structures: a *similarity-based inverted index* (named *SII*) of the target and a *pattern index* extracted from *SII* on the basis of the pattern labels. Through *SII*, nodes in the target with labels similar to those of the pattern are identified and organized in the levels in which they appear in the target. Fragments are generated by considering the ancestor-descendant relationships among such nodes. Then, identified fragments are combined in *regions*, allowing for the occurrence of nodes with labels not appearing in the pattern, as discussed before. Finally, some heuristics are employed to avoid considering all the possible ways of merging fragments into regions and for the efficient computation of similarity, thus making the approach more efficient.

More specifically, in the inverted index for a target, the labels appearing in the target are normalized with respect to the employed similarity relationship. Each label is then associated with the list of nodes labeled by the same or a similar label, ordered relying on the pre-order rank. The pattern index is a structure organized in levels, with an entry for each node in the pattern. The number of levels of the index depends on the levels in the tree in which nodes occur with labels similar to those in the pattern. For each level, nodes are ordered according to the pre-order rank. Fragments are generated through a visit of the pattern index. Each node in the first level of this index is the root of a fragment because of the way the index is built. Possible descendants of the node can be identified in the underlying levels. Given a generic level l of the pattern index, a node v can be a root of a fragment if and only if for none of the nodes u in previous levels, v is a descendant of u . If v is a descendant of a set of nodes U , v can be considered the child of the node $u \in U$ s.t. the distance between v and u is minimal.

A brute-force strategy for obtaining the best k results of a pattern query is to generate all results, sort them by score and return the k results with the highest score. A slight refinement saves space by using a heap to keep only the best k results found so far. The search for more efficient algorithms is constrained by the requirements of supporting a class of scoring functions as broad as possible, thus not necessarily monotone. Top- k algorithm deriving from Fagin's TA [16], indeed, requires the monotonicity of the aggregation function. Top- k processing with arbitrary ranking functions has been investigated in [47] with the restriction that the function is lower-bounded. An index-merge framework that performs progressive search over a space of states composed by joining index nodes is proposed. A state in this space is composed by joining multiple index nodes. Promising search states, in terms of their scores, are progressively materialized, while the states with no chances to yield the top- k answers are early-pruned. That approach has been used for top- k query processing in ArHeX [38].

6.6 Conclusion and Discussion

In the chapter, the main issues and some alternative approaches to deal with queries on XML documents accounting for approximation both on content and on structure

have been discussed. Among the devised approaches, a single preferable approach, working best than the others, cannot be identified, since this deeply depends on the characteristics of the XML collection on querying.

The characteristics of data in the collection on querying may suggest the measure to apply. The use of entropy-based measures to evaluate the level of heterogeneity of different aspects of documents is proposed in [36]. This allows to quantify the degrees of vocabulary and structural heterogeneity in a document collection. More in general, an analysis of the degrees of heterogeneity characterizing a data collection would allow to mine heterogeneity patterns that support the choice of the most appropriate similarity measure and the most effective approximate querying approach on the collection. For instance, it is not worth considering approximate tag matches at nodes, if the collection is very homogeneous at the vocabulary level. For what concerns the ranking, an interesting opportunity is to consider data for the selection of the measure to be employed, that is, to follow an *exploration-based* approach. In this case, data characteristics are considered to point out the measure (or the features of the measure) that should return the best result. By extracting different kinds of information from data (e.g., element tags, relevant words in element content, attributes, links), the features of the similarity measures to be applied on data can be revealed. For example, if the structure of the documents is highly regular, a content-based measure is expected to be more useful.

However, in contexts in which several collections of XML documents, stored at autonomous sources, are interactively queried, through a web application, it is not reasonable to fully analyze the documents available in the sources to extract enough information on their heterogeneity degrees to obtain the best similarity function to be used in approximating queries over the sources. The most effective approach in this context could be to start querying the sources with a starting similarity function and approximation approach and then use feedbacks on query execution to tune the approximation and similarity functions to be used in subsequent queries. Thus, in processing the subsequent queries submitted by the same user, that refine, adjust, or completely modify the previous one, the application will rely on feedbacks from previous query executions. For instance, feedbacks can reveal that a certain tag or a certain data content were matched exactly in the exploited source, thus there is no need to approximate conditions involving them when evaluated against data in that source. Similarly, one of the components of a composite measure may reveal not to contribute significantly to the overall score, and thus can be omitted from the measure. This can be seen as an instance of inter-query adaptive query processing [15] and can be realized by means of an appropriate Monitor-Assess-Plan-Execute architecture, through the collection of suitable information during query execution and their subsequent analysis. Another interesting direction could be to rely on user feedbacks on results to refine the approximation and the corresponding measure, in the same spirit of the approaches proposed in [12, 30].

References

1. Abiteboul, S., Buneman, P., Suciu, D.: *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann (1999)
2. Agrawal, S., Chaudhuri, S., Das, G., Gionis, A.: Automated Ranking of Database Query Results. In: CIDR (2003)
3. Amer-Yahia, S., Cho, S., Srivastava, D.: Tree Pattern Relaxation. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 496–513. Springer, Heidelberg (2002)
4. Amer-Yahia, S., Koudas, N., Marian, A., Srivastava, D., Toman, D.: Structure and Content Scoring for XML. In: VLDB, pp. 361–372 (2005)
5. Amer-Yahia, S., Lakshmanan, L.V.S., Pandit, S.: FleXPath: Flexible Structure and Full-Text Querying for XML. In: SIGMOD Conference, pp. 83–94 (2004)
6. Amer-Yahia, S., Lalmas, M.: XML Search: Languages, INEX and Scoring. SIGMOD Record 35(4), 16–23 (2006)
7. Augsten, N., Barbosa, D., Böhlen, M.H., Palpanas, T.: TASM: Top-k Approximate Subtree Matching. In: ICDE, pp. 353–364 (2010)
8. Augsten, N., Böhlen, M.H., Dyreson, C.E., Gamper, J.: Approximate Joins for Data-Centric XML. In: ICDE, pp. 814–823 (2008)
9. Augsten, N., Böhlen, M.H., Gamper, J.: Approximate Matching of Hierarchical Data Using pq-Grams. In: VLDB, pp. 301–312 (2005)
10. Baeza-Yates, R.A., Ribeiro-Neto, B.A.: *Modern Information Retrieval*. ACM Press / Addison-Wesley (1999)
11. Bruno, N., Koudas, N., Srivastava, D.: Holistic Twig Joins: Optimal XML Pattern Matching. In: SIGMOD Conference, pp. 310–321 (2002)
12. Cao, H., Qi, Y.Q., Candan, K.S., Sapino, M.L.: Feedback-driven Result Ranking and Query Refinement for Exploring Semi-structured Data Collections. In: EDBT, pp. 3–14 (2010)
13. Chaudhuri, S., Ramakrishnan, R., Weikum, G.: Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In: CIDR, pp. 1–12 (2005)
14. Damiani, E., Lavarini, N., Marrara, S., Oliboni, B., Pasini, D., Tanca, L., Viviani, G.: The APPROXML Tool Demonstration. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 753–755. Springer, Heidelberg (2002)
15. Deshpande, A., Ives, Z.G., Raman, V.: Adaptive Query Processing. *Foundations and Trends in Databases* 1(1), 1–140 (2007)
16. Fagin, R., Lotem, A., Naor, M.: Optimal Aggregation Algorithms for Middleware. *J. Comput. Syst. Sci.* 66(4), 614–656 (2003)
17. Fellbaum, C.: *WordNet: An Electronic Lexical Database*. MIT Press (1998)
18. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: VLDB, pp. 436–445 (1997)
19. Gou, G., Chirkova, R.: Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Trans. Knowl. Data Eng.* 19(10), 1381–1403 (2007)
20. Grust, T., van Keulen, M., Teubner, J.: Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In: VLDB, pp. 524–525 (2003)
21. Guerrini, G., Mesiti, M., Bertino, E.: Structural Similarity Measures in Sources of XML Documents. In: Darmont, J., Boussaid, O. (eds.) *Processing and Managing Complex Data for Decision Support*, pp. 247–279. IDEA Group (2006)

22. Guerrini, G., Mesiti, M., Sanz, I.: An Overview of Similarity Measures for Clustering XML Documents. In: Vakali, A., Pallis, G. (eds.) *Web Data Management Practices: Emerging Techniques and Technologies*, IDEA Group (2007)
23. Guha, S., Jagadish, H.V., Koudas, N., Srivastava, D., Yu, T.: Approximate XML Joins. In: *SIGMOD Conference*, pp. 287–298 (2002)
24. Hung, E., Deng, Y., Subrahmanian, V.S.: TOSS: An Extension of TAX with Ontologies and Similarity Queries. In: *SIGMOD Conference*, pp. 719–730 (2004)
25. Ide, N., Véronis, J.: Introduction to the Special Issue on Word Sense Disambiguation: The State of the Art. *Computational Linguistics* 24(1), 1–40 (1998)
26. Ilyas, I.F., Beskales, G., Soliman, M.A.: A Survey of Top- k Query Processing Techniques in Relational Database Systems. *ACM Comput. Surv.* 40(4) (2008)
27. Jones, K.S., Walker, S., Robertson, S.E.: A Probabilistic Model of Information Retrieval: Development and Comparative Experiments - Part 1 and Part 2. *Inf. Process. Manage.* 36(6), 779–840 (2000)
28. Lalmas, M.: *XML Retrieval. Synthesis Lectures on Information Concepts, Retrieval, and Services*. Morgan & Claypool Publishers (2009)
29. Lalmas, M., Trotman, A.: XML Retrieval. In: *Encyclopedia of Database Systems*, pp. 3616–3621 (2009)
30. Lau, H.L., Ng, W.: A Multi-Ranker Model for Adaptive XML Searching. *VLDB J.* 17(1), 57–80 (2008)
31. Lovins, J.B.: Development of a Stemming Algorithm. *Mechanical Translation and Computational Linguistics* 11, 22–31 (1968)
32. Marian, A., Amer-Yahia, S., Koudas, N., Srivastava, D.: Adaptive Processing of Top- k Queries in XML. In: *ICDE*, pp. 162–173 (2005)
33. Marian, A., Schenkel, R., Theobald, M.: Ranked XML Processing. In: *Encyclopedia of Database Systems*, pp. 2325–2332 (2009)
34. Navarro, G.: A Guided Tour to Approximate String Matching. *ACM Comput. Surv.* 33(1), 31–88 (2001)
35. Salton, G., McGill, M.J.: *Introduction to Modern Information Retrieval*. McGraw-Hill (1983)
36. Sanz, I., Llavori, R.B., Mesiti, M., Guerrini, G.: ArHeX: Flexible Composition of Indexes and Similarity Measures for XML. In: *ICDE Workshops*, pp. 281–284 (2007)
37. Sanz, I., Mesiti, M., Guerrini, G., Llavori, R.B.: Fragment-Based Approximate Retrieval in Highly Heterogeneous XML Collections. *Data Knowl. Eng.* 64(1), 266–293 (2008)
38. Sanz, I., Mesiti, M., Guerrini, G., Llavori, R.B.: Flexible Multi-Similarity XML Data Querying with Top- k Processing. Tech. rep., Universitat Jaume I (2009)
39. Schlieder, T.: Schema-Driven Evaluation of Approximate Tree-Pattern Queries. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) *EDBT 2002. LNCS*, vol. 2287, pp. 514–532. Springer, Heidelberg (2002)
40. Tai, K.C.: The Tree-to-Tree Correction Problem. *J. ACM* 26(3), 422–433 (1979)
41. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and Querying Ordered XML using a Relational Database System. In: *SIGMOD Conference*, pp. 204–215 (2002)
42. Tekli, J., Chbeir, R., Yé tongnon, K.: An Overview on XML Similarity: Background, Current Trends and Future Directions. *Computer Science Review* 3(3), 151–173 (2009)
43. Theobald, M., Bast, H., Majumdar, D., Schenkel, R., Weikum, G.: TopX: Efficient and Versatile Top- k Query Processing for Semistructured Data. *VLDB J.* 17(1), 81–115 (2008)

44. W3C: XML Path Language (XPath) 2.0 (2007), <http://www.w3.org/TR/xpath20/>
45. W3C: XQuery 1.0: An XML Query Language (2007), <http://www.w3.org/TR/xquery/>
46. W3C: XQuery and XPath Full Text 1.0 (2010), <http://www.w3.org/TR/xpath-full-text-10/>
47. Xin, D., Han, J., Chang, K.C.C.: Progressive and Selective Merge: Computing Top- k with ad-hoc Ranking Functions. In: SIGMOD Conference, pp. 103–114 (2007)
48. Zhang, K., Shasha, D.: Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18(6), 1245–1262 (1989)

Chapter 7

Progressive and Approximate Join Algorithms on Data Streams

Wee Hyong Tok and Stéphane Bressan

Abstract. In this chapter, we discuss the design and implementation of join algorithms for data streaming systems, where memory is often limited relative to the data that needs to be processed. We first focus on progressive join algorithms for various data models. We introduce a framework for progressive join processing, called the Result Rate based Progressive Join (RRPJ) framework which can be used for join processing for various data models, and discuss its various instantiations for processing relational, high-dimensional, spatial and XML data.

We then consider progressive and approximate join algorithms. The need for approximate join algorithms is motivated by the observation that users often do not require complete set of answers. Some answers, which we refer to as an approximate result, are often sufficient. Users expect the approximate result to be either the largest possible or the most representative (or both) given the resources available. We discuss the tradeoffs between maximizing quantity and quality of the approximate result. To address the different tradeoffs, we discuss a family of algorithms for progressive and approximate join processing.

7.1 Introduction

The emergence of ubiquitous network connectivity allows data to be delivered as streams. The data on these streams can come in a variety of data models, from relational to spatial, high-dimensional and XML. New applications (e.g., sensors databases, P2P, cloud computing, XML aggregators or data exploration, see the

Wee Hyong Tok
Microsoft
e-mail: weetok@microsoft.com

Stéphane Bressan
School of Computing, National University of Singapore
e-mail: steph@comp.nus.edu.sg

CONTROL system [18]) that want to leverage these data need to be able to progressively process data from these streams. Progressive query processing techniques produce results as new data arrives. These query processing techniques are suited to the processing of queries to data streams. They need to be designed under two main constraints. Firstly, because of the hardware constraints of mobile devices or/and the massive amounts of data that needs to be processed, the size of main memory is limited relative to the data. Secondly, data delivery is unpredictable.

Joins are one of the main building blocks of a query processing system. In this chapter, we focus on the design and implementation of progressive and approximate join algorithms for data stream applications. The algorithms that we will explore in this chapter work with limited memory and adapt to unpredictable data delivery rate.

We first present and evaluate a generic progressive join processing framework, called Result Rate-Based Progressive Join framework (RRPJ), that can deliver results incrementally using limited memory. In order to demonstrate the generic nature of the proposed framework, we propose four instantiations of the framework for relational, spatial, high-dimensional and XML join processing. We show that RRPJ yields good performance compared to other state-of-art progressive join algorithms for the various data models.

We then present and evaluate join algorithms that are not only progressive but also approximate. Often, users do not require complete set of answers. Some answers, which we refer to as an approximate result, are often sufficient. Given limited resources, users expect the approximate result to be either the largest possible or the most representative (or both). We discuss the tradeoffs between maximizing quantity and quality of the approximate result. To address the different tradeoffs, we present and evaluate a family of algorithms for progressive and approximate join processing.

7.2 Background

Many join processing algorithms [47, 46, 16, 14, 38, 32, 51, 40, 26, 27] have been proposed. Most of these algorithms focus on the equi-join. In order to ensure that join processing is non-blocking (or progressive), many of these equi-join algorithms leverage the seminal work on symmetric hash join's (SHJ) [49]. SHJ assumes the use of in-memory hash tables, and makes use of an insert-probe paradigm. This allows results to be delivered progressively to users. In an insert-probe paradigm, a newly-arrived tuple is first used to probe the hash partition for the corresponding data stream. If there are matching tuples (based on the join predicate), the matching tuples are output as results. The newly-arrived tuple is then inserted into its own hash partition. This allows results to be output immediately whenever new tuples arrive.

In order to address the issue of limited memory, many subsequently proposed progressive relational join algorithms (e.g [47, 46, 32, 40]) considered an extension of the SHJ model, where both in-memory and disk-based hash partitions are

used. The extended version of the SHJ model consists of three phases: (1) Active (2) Blocked (3) Cleanup. In the active phase, data is continuously arriving from the data streams. Whenever a newly tuple arrives, it is first used to probe the hash partitions for the corresponding data stream, before it is inserted into its own hash partitions. Whenever memory is full, some of the in-memory tuples are flushed to disk to make space for new-arriving tuples. Whenever the data stream blocks, the extended SHJ moves into a blocked phase. During the blocked phase, data from the disk partitions are retrieved and joined with either in-memory or disk-resident tuples from the corresponding data streams. This allows the delays from the blocked data streams to be hidden from the end user. In the cleanup phase, all tuples that have not been previously joined are then joined to ensure that the results produced are completed.

In order to maximize result throughput, a key focus of existing progressive join algorithms is to determine the set of tuples that are flushed to disk whenever memory is full. Many flushing techniques have been proposed for progressive join algorithms. These techniques can be classified as heuristic-based or statistics-based. In heuristic-based techniques, a set of heuristics govern the selection of tuples or partitions to be flushed to disk. These heuristics ranges from flushing the largest (e.g., XJoin [46]) to a concurrent flushing of partitions (e.g., Hash-Merge Join (HMJ) [32]). In statistics-based techniques, a statistical model is maintained on the input distribution. Whenever a flushing decision needs to be made, the statistical model can be used to determine the tuples or partitions that are least likely to contribute to a future result. These tuples or partitions are then flushed to disk. Amongst the various statistical based techniques, the Rate-based Progressive Join (RPJ) and Locality-Aware (LA) models are the state-of-art in statistic-based progressive join algorithms. RPJ rely on the availability of an analytical model deriving the output probabilities from statistics on the input data. This is possible in the case of relational equi-joins but embeds some uniformity assumptions that might not hold for skewed datasets. For example, if the data within each partition is non-uniform, the RPJ local uniformity assumption is invalid.

Consider the two partitions, belonging to dataset R and S respectively, presented in Figure 7.1. The grayed area represent the data and white an empty space. The vertical axis for the rectangles represents data values. Suppose in both Figure (a) and (b), N tuples have arrived. In Figure 7.1(a), the N tuples is uniformly distributed across the entire partitions of each dataset. Whereas in Figure 7.1(b), the N tuples is distributed within a specific numeric range (i.e., areas marked grey). Assume the same number of tuples have arrived for both cases, then $P(1|R)$ and $P(1|S)$ would be the same. However, it is important to note that if partition 1 is selected to be the partition to be kept in memory, the partitions in Figure 7.1(a) would produce results as predicted by RPJ. Whereas the partitions in Figure 7.1(b) would fail to produce any results. Though RPJ attempts to amortize the effect of historical arrivals of each relation, it assumes that the data distribution remains stable throughout the lifetime of the join, which makes it less useful when the data distribution are changing (which is common in long-running data streams).

The LA model is designed for approximate sliding window join on relational data. It relies on determining the inter-arrival distance between tuples of similar values in order to compute the utility of the tuple. Consequently, the utility of the tuple is used to guide the tuples to be flushed to disk. In the case of relational data, a similar tuple could be one that has the same value with a previous tuple. However, for non-relational data, such as spatial or high-dimensional data, the notion of similarity becomes blurred. Another limitation of the LA model is that it is unable to deal with changes in the underlying data distribution. This is because with a frequently changing data distribution, which is common in long running data streams, the reference locality, which is a central concept in the LA model cannot be easily computed. Hence, both RPJ and LA model cannot be easily extended to deal with non-relational data models.

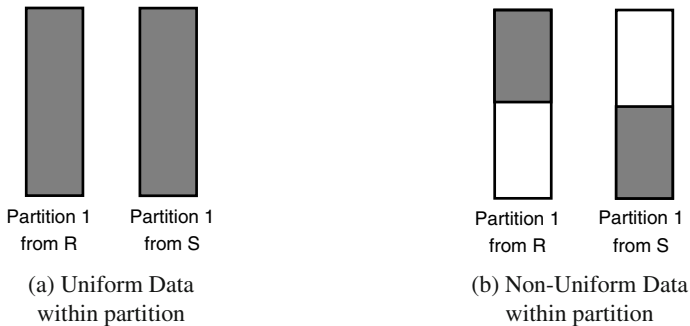


Fig. 7.1 Data in a Partition.

7.3 Why Progressive Joins?

As many of the existing progressive join techniques are designed for relational data model, they are not easily adapted to other data models. As a result, new progressive join techniques, with different flushing policies need to be proposed for each type of data that needs to be processed. In addition, when processing large datasets or data streams, the amount of memory available for keeping the data is often limited. Whenever memory is full, a flushing policy is used to determine the data that are either flushed to disk partitions, or discarded. Data are flushed to disk partitions if the user is interested in the complete production of results. On the other hand, if the user is interested in approximate results, some of the in-memory data can be discarded.

This research is driven by the need to design a generic, progressive join framework that meets three objectives. Firstly, the framework must be easily generalized to different data models (e.g., relational, spatial, high-dimensional XML). Secondly, the progressive join framework must work with limited memory. Thirdly, it is important to identify the metrics that are suitable for evaluating the performance of the progressive joins. The chapter is divided into the following parts.

The first part of the chapter presents motivation for a generic progressive join framework, which can be used in different data models. In order to address all these issues, we focus on SHJ-based algorithms as one of the building blocks for designing a progressive join framework. This is because the probe-insert paradigm used in SHJ-based algorithms provides the basis for producing results (if any) whenever data is available. As SHJ-based algorithms rely on hashing for probing and insertion, the challenge is to identify the appropriate hash-based data structure for each of the data models. In order to deal with limited memory, the flushing policy is one of the key factors for maximizing the result throughout or the quality of the approximate result subset produced. Most importantly, the flushing policy must be independent of the data model. While heuristics-based flushing policies meet the criteria of data model independence, they perform poorly compared to the statistics-based techniques. Most importantly, statistics-based techniques provide strong theoretical guarantees on the expected result output. However, existing statistic-based techniques suffer from the data model dependence. While many good statistic-based techniques have been proposed for the relational data model, none of these can be easily extended for other data models. In order to have a generic flushing policy, we observed that the goal of progressive join algorithms is on result throughput maximization. Motivated by this, we conjectured that the statistics used to determine the data that are flushed from memory should be result-driven.

The second part of the chapter is motivated by the observation that users might not need the production of complete results. Also, in data stream applications, the notion of complete results is impractical, since the data streams can be potentially infinite. When approximate results are produced, it is important to distinguish between the quantity and quality of the results. Noting that sampling-based techniques have been previously disqualified by the authors of [14] without further investigations, we show that this disqualification is mistaken. In the chapter, we show that a stratified sampling approach is both effective and efficient for progressive, approximate join processing.

7.4 Joins from Different Data Models Flock Together

7.4.1 Relational Joins

Many methods [46, 32, 40, 27] have been proposed to deal with the progressive equi-join problem on relational data streams. A recent trend amongst these methods is to make use of probabilistic models on the data distribution to determine the best data to be kept in memory.

RPJ [40] is a multi-stage hash-based join algorithm, for joining data that are transmitted from remote data sources over a unreliable network. Similar to hash-based join algorithms like XJoin, RPJ stores the data into partitions. Each partition consists of two portions, one residing in memory and the other on disk. Whenever a new data arrives, RPJ computes the hash value based on the join attribute, and uses this to probe the corresponding partition to identify matching tuples. The RPJ algorithm

consists of several stages. The stages are as follows: (1) Memory-to-Memory (2) Reactive. In the Memory-to-Memory stage (mm-stage), arriving data are joined with the in-memory tuples from the other data set. Whenever the memory overflows, selected tuples are flushed to the disk partitions. The Reactive Stage is triggered whenever the data source blocks. It consists of two sub-tasks: (i) Memory-Disk (Md-task) and (2) Disk-Disk (Dd-task). In the Md-task, data that are in memory are joined with their corresponding partitions on disk. And in the Dd-task, data that are on disk are joined with the corresponding partitions from the other data sets on disk. One of the key ideas of RPJ is to maximize the number of results tuples by keeping tuples that have higher chances of producing results with the tuples from the corresponding data set in memory. An *Optimal Flush* technique flushes tuples that are not useful to disk. This is achieved by building a model on the tuples' arrival pattern and data distribution. Whenever memory becomes full, the model can be used to determine probabilistically which tuples are least likely to produce tuples with the other incoming data, and hence flushed from memory to disk. RPJ computes $p_i^{arr}(v)$, which denotes the probability that the next incoming tuple would be from data source i , and has the value v . Using the arrival probabilities, the RPJ strategy is illustrated by the following example. The tuples from two remote data sources R and S , are continuously retrieved, and joined. The join condition is $R.a = S.a$, the domain of the join attribute, a , is $\{2,4,6,8\}$. The arrival probabilities for R are: $p_R^{arr}(2) = 10\%$, $p_R^{arr}(4) = 15\%$, $p_R^{arr}(6) = 4\%$ and $p_R^{arr}(8) = 6\%$; whereas the arrival probabilities for S are: $p_S^{arr}(2) = 5\%$, $p_S^{arr}(4) = 20\%$, $p_S^{arr}(6) = 30\%$ and $p_S^{arr}(8) = 10\%$. At the instance when memory overflows, each of the data sources has 2 tuples for each value in memory. Suppose $n_{flush}=6$ tuples need to be flushed from memory. Since the arrival probability for $p_R^{arr}(6) = 4\%$ is the smallest, we will need to flush 2 S -tuples with the value 6 from memory (i.e., these S -tuples would be least likely to produce results since the corresponding R -tuples do not arrive as often compared to other tuples). Since $n_{flush}=6$, we would need to flush 4 more tuples from memory. We consider the next smallest arrival probability. In this case, $p_S^{arr}(2) = 5\%$ is the smallest. Thus, we flush 2 R -tuples with the value 2 from memory. Finally, we consider $p_R^{arr}(8) = 6\%$, and flush 2 S -tuples with the value 8 from memory.

[27] observes that a data stream exhibits reference locality when tuples with specific attribute values has a higher probability of re-appearing in a future time interval. Leveraging this observation, the Locality-Aware (LA) model exploits the reference locality caused by both long-term popularity and short-term correlations. This is described by the following model: $x_n = x_{n-i}$ (with probability a_i); $x_n = y$ (with probability b , where $1 \leq i \leq h$ and $b + \sum_{i=1}^h a_i = 1$). y denotes a random variable that is independent and identically distributed (IID) with respect to the probability distribution of the popularity, P . Using this model, the probability that a tuple t will appear at the n -th position of the stream is given by $Prob(x_n = t | x_{n-1}, \dots, x_{n-h}) = bP(t) + \sum_{j=1}^h a_j \delta(x_{n-j}, t)$ ($\delta(x_k, t) = 1$ if $x_k = t$, and it is 0 otherwise). Using the LA model, the marginal utility of a tuple is then derived, and is then used as the basis for determining the tuples to be flushed to disk whenever memory is full.

7.4.2 Spatial Joins

In this section, we discuss various types of spatial join processing techniques that have been proposed. In addition, we have also conducted an extensive survey on continuous query processing on spatial data, which is presented in [21].

Spatial index structures such as R-tree [17], R+-tree [35], R*-tree [4] and PMR quad-tree [33] were commonly used together with spatial joins. In [10], Brinkhoff et al. proposed a spatial join algorithm which uses a depth-first synchronized traversal of two R-trees. The implicit assumption is that the R-trees has already been pre-constructed for the two spatial relations to be joined. A subsequent improvement to the synchronized traversal was proposed by [20], called *Breadth First R-tree Join* (BFRJ). By traversing the R-tree level by level, BFRJ was able to perform global optimization on which are the next level nodes to be accessed, and hence minimize page faults. In [29], a seeded tree method for spatial joins was proposed. It assumes that there is a pre-computed R-tree index for one of the spatial relations. The initial levels of the R-tree are then used to provide the initial levels (i.e., *seeds*) for the dynamically constructed R-tree of the corresponding spatial relation. An integrated approach for handling multi-way spatial join was proposed in [31]. Noting that the seeded tree approach performs poorly when the fanout of the tree is large to fit into a small buffer small, [31] also proposed the *Slot Index Spatial Join* to tackle the problem.

The use of hashing was explored in [30, 34]. In [30], the Spatial Hash Join (SHJ) was proposed to compute the spatial join for spatial data sets which has no indexes pre-constructed. Similar to its relational counter-part, the spatial hash join consists of two phases: (1) Partitioning Phase and (2) Join Phase. In the Partitioning Phase, a spatial partitioning function first divides the data into outer and inner buckets. In order to address issues due to the *coherent assignment* problem, a multiple assignment of data into several buckets was adopted. This allows two bucket pairs to be matched exactly once, and reduces the need to scan other buckets. In the join phase, the inner and outer buckets are then joined to produce results. The Partition Based Spatial-Merge (PBSM) method proposed in [34] first divides the space using a grid with fixed-sizes cells (i.e., tiles). These tiles are then mapped to a set of partitions. The data objects in the partitions are then joined using a computational geometry based plane-sweeping approach. In [1], the Scalable Sweeping-Based Spatial Join (SBSJ) was proposed. The design of SBSJ is based on the observation that in plane-sweeping approaches, only the objects that intersect with the sweeping line need to be in memory. In addition, SBSJ ensures that data is not replicated.

Spatial join algorithms based on other novel data structures have also been proposed. The *Filter Trees* [36], a multi-granularity hierarchical structure, was used as an alternative to R-trees and its variants. Noting that techniques such as PBSM and SHJ requires replication of data, the Size Separation Spatial Join (S³J) [23] was proposed by building incomplete *Filter Trees* on-the-fly and using them in join processing.

Existing spatial join processing techniques focus on reducing the number of I/Os for datasets that reside locally. None of these proposed techniques are optimized for

delivering the initial results quickly, and do not consider the case where spatial data are continuously delivered from remote data sources.

7.4.3 High-Dimensional Distance-Similarity Joins

Many efficient distance similarity joins [37, 25, 6, 7, 22] have been proposed for high-dimensional data. To facilitate efficient join processing, similarity join algorithms often relies on spatial indices. R-trees (and variants) [17], X-tree [5] or the ϵ -kdb tree [37] are commonly used. The Multidimensional Spatial Join (MSJ) [25, 24] sorts the data based on their Hilbert values, and uses a multi-way merge to obtain the result. The Epsilon Grid Order (EGO) [7] orders the data objects based on the position of the grid-cells. Another related area is the K-nearest Neighbor (KNN) [8, 9]. The Multi-page Index (MUX) method [9], uses R-trees to reduce the CPU and I/O costs of performing the KNN join. GORDER [50] uses Principal Component Analysis (PCA) to identify key dimensions, and uses a grid for ordering the data.

The main limitation of conventional distance similarity join algorithms is that they are designed mainly for datasets that reside locally. Hence, they are not able to deliver results progressively.

7.4.4 Progressive XML Structural Joins

[19] proposed a Massively Multi-Query Join Processing (MMQJP) technique for processing value joins over multiple XML data streams. Similar to our approach, MMQJP consists of two phases: XPath Evaluation and Join Processing phase. In the XPath evaluation phase, the XML data streams are matched and auxiliary information stored as relations in a commercial database management systems (DBMS) - Microsoft SQL Server. The auxiliary information are then used during the join processing phase for computing results. Thus, MMQJP can only deliver results when the entire XML documents have arrived. In addition, MMQJP have no control over the flushing policy due to its dependence on the commercial DBMS. In contrast to MMQJP, our proposed technique delivers results progressively as portions of the streamed XML documents arrived.

In addition, a physical algebra for XQuery was proposed in [39]. The algebra allows XML streaming operators to be intermixed with conventional XML and relational operators in a query plan. This allows pipelined plans to be easily defined. The work in [39] does not consider memory management issues.

7.5 Generic Progressive Join Framework

In this chapter, we present the issues that need to be considered for designing a generic progressive join framework. These include the need for a data structure

to support efficient frequent probe-insert, and a data-model independent flushing policy to determine the data to be flushed to disk whenever memory is full.

7.5.1 Building Blocks for Generic Progressive Join Framework

7.5.1.1 Data Structures

We focus on data structures used for hash-based joins. It is important to note that even though progressive join algorithms based on sort-merge paradigm (e.g., [16]) exists. These algorithms are not able to deliver initial results quickly, as results can only be produced when the data structure (i.e., sweep area) used is sufficiently full before sorting can be performed.

A data structure, D , used to store data and support the join algorithm must have the following required properties: (1) Correctness and (2) Completeness. It must ensure that the results produced are correct with respect to the join predicate used. In addition, it must ensure that the complete result set can be produced. A desirable property is having a *minimal* data structure. This means that the data structure must ensure that the minimum number of partitions are scanned in order to compute the complete result set. For example, given a data structure, D . D is used for storing data from two data sources R and S . Given a tuple t from R , if all the partitions from S need to be scanned in order to identify the result set, then D is not minimal.

D divides the data space into equal-sized partitions, each denoted by P_i , where i denotes the i -th partition. Whenever a new data object o arrives, a partitioning function f determines the partition which o belongs to. Formally, $f(o) \rightarrow I$, where I denotes a set of partitions, N denotes the total number of partitions, and $\{1, \dots, N\} \in I$. Ideally, $I = 1$ (i.e each object is assigned to a single partition).

For relational data, this can be easily achieved by choosing a good hash function, f , which assigns each data object into a single partition. For spatial data, we make use of the same spatial partitioning function used in Spatial Hash Joins [30]. Each spatial object is replicated into the grid-cells in which it intersects. As observed in [30], the replication is necessary to allow pairwise joins between partitions from each data source. For high-dimensional data (i.e., n -dimension), each object is inserted into the partition (i.e., grid-cell) in which it falls into.

Next, we introduce the notion of a correspondence function, κ . κ maps a partition P to the set of partitions from the other data stream that need to be scanned. Formally, $\kappa(P) \rightarrow J$, where J denotes the set of partition(s) that need to be scanned from the other data stream. This is illustrated in Figure 7.2, where a partition P is mapped to a partition in the other data grid. It is important to note that it is possible that a partition from one grid need not necessary map to the same partition in the other data grid.

For both relational data and spatial data, κ is usually the identity correspondence (i.e., $I = J$), which is necessary to ensure that only pairwise partitions (one from each of the data streams) are scanned in order to identify the complete result set. This help to prevent redundant scanning of partitions which will not yield any results.

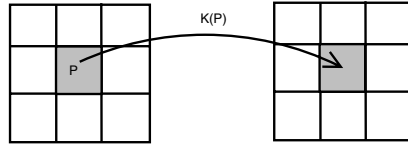


Fig. 7.2 Correspondence Function, κ .

If the partitioning functions used for each of the data stream are not the same, then κ is non-identity. For high-dimensional data, κ is non-identity. This is because, for each grid-cell from one data stream, the grid-cell that contains data that are epsilon-distance needs to be scanned in order to find the complete result set. Table 7.1 summarizes the data structure, and the partitioning function used for each data type.

Table 7.1 Various Data Structures.

Data Type	Data Structure, DS	Partitioning Function, f	$ I $	κ
Relational	Hash-based partitions	Modulo	1	Identity
Spatial	2-dimensional Grid	Insert each object into the grid-cells in which it intersects	≥ 1	Identity
High Dimensional Data	n-dimensional Grid	Insert each object into the grid-cell it falls into	1	Non-identity

7.5.1.2 Flushing Policy

Whenever memory becomes full, the flushing policy determines the tuples to be flushed to disk. The goal of the generic progressive join framework is to design flushing policies that are independent of the data model used. Consequently, this allows the generic progressive join framework to be easily instantiated for other data models easily. In contrast, flushing policies which are dependent on the input data distribution and the type of join predicates cannot be easily generalized.

7.5.2 Progressive Join Framework

We consider the problem of performing a join J between two datasets R and S , which are transmitted from remote data sources through an unpredictable network. Let R and S be denoted by $R = \{r_1, r_2, \dots, r_n\}$, and $S = \{s_1, s_2, \dots, s_m\}$, where r_i and s_j denotes the i -th and j -th data objects. The join predicate is denoted by J_{pred} . Formally, (r_i, s_j) is reported as the result if r_i and s_j satisfies J_{pred} . The goal is to deliver initial results quickly and ensure a high result-throughput.

The general form of a progressive join algorithm presented in Algorithm 1. In Algorithm 1 we assume that there are two remote data sources, R and S . The in-memory data structures used to store the data objects from R and S are denoted by

DR and DS respectively. $endOfStream(\dots)$ determines whether data from the stream has completely arrived. This is usually indicated by an end-of-stream marker sent by the remote data source. $isBlocked(\dots)$ determines whether data from the stream is blocked (i.e., data did not arrive for a user-defined duration). $ProcessUnJoinedData()$ determines the data that has not been previously joined and joins them to produce results. $select(R,S)$ gets the data from either of the data streams to be processed.

Algorithm 1. Generic Progressive Join.

```

1: while ( !endOfStream(R) and !endOfStream(S) ) do
2:   if ( isBlocked(R) and isBlocked(S) ) then
3:     //Blocking Phase
4:     ProcessUnJoinedData()
5:   end if

6:   //In-memory Phase
7:   tuple  $t = select(R,S)$ 

8:   if (  $t.src == R$  ) then
9:      $DS.probe(t)$ 
10:     $DR.insert(t)$ 
11:  else if (  $t.src == S$  ) then
12:     $DR.probe(t)$ 
13:     $DS.insert(t)$ 
14:  end if
15: end while

16: //Cleanup Phase
17:  $CleanUp()$ 

18: return (Results tuples from the join)

```

In the In-Memory phase, whenever a new data object t arrives, it is used to probe (line 9 or line 12) the corresponding data structure to identify all the data objects in DS_s that joins with it. Once the probe is completed, t is then inserted the data structure used to store the in-memory data (line 10 or line 13). During the insertion of t , the algorithm needs to check whether the memory is full. If it is full, data needs to be flushed to disk. This is determined by a flushing policy.

When both the data sources block (lines 2-5), the algorithm moves into the *Blocking Phase*. In order to produce results during this phase, the join algorithm joins the in-memory data with the on-disk data. When all the in-memory data has been joined, the algorithm would need to join disk-resident data from both the data sources. This allows results to be produced even though both data streams are blocked.

In the Cleanup phase (line 17), data which have not been joined in the prior phases are joined. These include joining in-memory data with disk-resident data and disk-resident data with disk-resident data. These ensure that the complete result set is produced. It is important to that due to the multiple invocations of the various phases,

duplicate results would be produced. These duplicates are removed using online duplicate elimination methods which has been extensively described in [46] and [40].

7.5.2.1 Result-Rated Based Flushing

In this section, we present a flushing policy which maintains statistics over the result distribution, instead of the data distribution. This is motivated by the fact that in most progressive join scenarios, we are concerned with delivering initial results quickly and maintaining a high overall throughput. Hence, the criteria used to determine the tuples that are flushed to disk whenever memory becomes full should be ‘result-motivated’. We refer to join algorithms that make use of the result-rate based flushing policy as Result-Rate Based Progressive Join (RRPJ).

Whenever memory is full, we compute the Th_i values (i.e. value computed by formula given in Equation 7.3) for all the partitions. Partitions with the lowest Th_i values will then be flushed to disk, and the newly arrived tuple inserted. The main difference between the RRPJ flushing and RPJ is that the Th_i values are reflective of the output (i.e., results) distribution over the data partitions. In contrast, the RPJ values are based on the input data distribution.

To compute the Th_i values (computed using Equation 7.3), we track the total number of tuples, n_i (for each partition), that contribute to a join result from the probes against the partition. Intuitively, RRPJ tracks the *join throughput* of each partition. Whenever memory becomes full, we flush n_{flush} (user-defined parameter) tuples from the partition that have the smallest Th_i values, since these partitions have produced the least result so far. If the number of tuples in the partition is less than n_{flush} , we move on to the partition with the next lowest Th_i values.

Given two timestamps t_1 and t_2 ($t_2 > t_1$) and the number of join results produced at t_1 and t_2 are n_1 and n_2 respectively. A straightforward definition of the throughput of a partition i , denoted by Th_i , is given in Equation 7.1

$$Th_i = \frac{n_2 - n_1}{t_2 - t_1} \text{ (version 1)} \quad (7.1)$$

From Equation 7.1, we can observe that since $(t_2 - t_1)$ is the same for all partitions, it suffices to maintain counters on just the number of results produced (i.e., n_1 and n_2). A partition with a high Th_i value will be the partition which has higher potential of producing the most results. Moreover, it is important to note that Equation 7.1 does not take into consideration the size of the partitions and its impact on the number of results produced. Intuitively, a large partition will produce more results. It is important to note that this might not always be true. For example, a partition might contain few tuples, but produces a lot of results. This partition should be favored over a relatively larger partition which is also producing the same number of results. Besides considering the result distribution amongst the partitions, we must also consider the following: (1) Total number of tuples that have arrived, (2) Number of tuples in each partition, (3) Number of result tuples produced by each partition and (4) Total results produced by the system. Therefore, we use an improved definition for Th_i , given below.

Suppose there are P partitions maintained for the relation. Let N_i denote the number of tuples in partition i ($1 \leq i \leq P$), and R_i denote the number of result tuples produced by partition i . Then, the Th_i value for a partition i can be computed. In Equation 7.2 we consider the ratio of the results produced to the total number of results produced so far (i.e., numerator), and also the ratio of the number of tuples in a partition to the total number of tuples that have arrived (i.e., denominator).

$$Th_i = \left(\frac{R_i}{\sum_{j=1}^P R_j} \right) / \left(\frac{N_i}{\sum_{j=1}^P N_j} \right) = \frac{R_i \times \sum_{j=1}^P N_j}{\sum_{j=1}^P R_j \times N_i} \quad (\text{version 2}) \quad (7.2)$$

Since the total number of results produced and the total number of tuples is the same for all partitions, Equation 7.2 can be simplified. This is given in Equation 7.3.

$$Th_i = \frac{R_i}{N_i} \quad (\text{version 2 - after simplification}) \quad (7.3)$$

Equation 7.3 computes the Th_i value w.r.t to the size of the partition. For example, let us consider two cases. In case (1), suppose $N_i = 1$ (i.e., one tuple in the partition) and $R_i = 100$. In case (2), suppose $N_i = 10$ and $R_i = 1000$. Then, the Th_i values for case (1) and (2) are the same. This prevents large partitions from unfairly dominating the smaller partitions (due to the potential large number of results produced by larger partitions) when a choice needs to be made on which partitions should be flushed to disk.

7.5.2.2 Amortized RRPJ (ARRPJ)

In order to allow RRPJ to be less susceptible to varying data distributions, we introduce Amortized RRPJ (ARRPJ). Suppose there are two partitions P_1 and P_2 , each containing 10 tuples. If P_1 produces 5 and 45 result tuples at timestamp 1 and 2 respectively, the Th_1 value is 5. If partition P_2 produces 45 and 5 result tuples at timestamp 1 and 2 respectively, the Th_2 value for P_2 will also be 5. From the above example, we can observe that the two scenarios cannot be easily differentiated. However, we should favor partition P_1 since it is obviously producing more results than P_2 currently. This is important because we want to ensure that tuples that are kept in memory are able to produce more results because of its current state, and not due to a past state.

To achieve this, let σ be a user-tunable factor that determines the impact of historical result values. The amortized RRPJ value, denoted as A_i^t , for a partition i at time t is presented in Equation 7.4. When $\sigma = 1.0$, then the amortized RRPJ is exactly the same as the RRPJ. When $\sigma = 0.0$, then only the latest RRPJ values are considered. By varying the values of σ between 0.0 to 1.0 (inclusive), we can then control the effect of historical RRPJ on the overall flushing behavior of the system.

$$A_i^t = \frac{\sigma^t r_i^0 + \sigma^{t-1} r_i^1 + \sigma^{t-2} r_i^2 + \dots + \sigma^1 r_i^{t-1} + \sigma^0 r_i^t}{N_i} = \frac{\sum_{j=0}^t \sigma^{(t-j)} r_i^j}{N_i} \quad (7.4)$$

7.5.3 RRPJ Instantiations

In the various instantiations of the framework, RRPJ is effective and efficient and is able to ensure a high result throughput using limited memory. An early version of the generic progressive join framework for spatial data, called *JGradient*, which builds a statistical model based on the result output is presented in [41]. Using the insights from [41], we propose a generic progressive framework, called Result-rate based progressive join (RRPJ) for relational data streams. RRPJ improves on *JGradient* in several aspects. Firstly, RRPJ take into consideration the size of each of the hash partitions. Secondly, an amortized version of RRPJ is introduced to handle changes in the result distribution from long-running data streams. The results of this research have been published in [44]. In order to show that the RRPJ can be instantiated for other data models, we studied the issues that arise from using the framework for high-dimensional data streams. We show that the high-dimensional instantiation, called *RRPJ High Dimensional* is able to maximize the results produced using limited memory. The results of this research have been published in [43].

The use of the the RRPJ framework for progressive XML value join processing is presented in [45]. In [45], For-Where-Return (FWR) XQuery queries are decomposed into a query plan that composes of twig queries and hash joins. In addition, using the result-oriented approach for query processing, a result-oriented method for routing tuples in a multi-way join, called Result-Oriented Routing (*RoR*) is also presented. *RoR* is used for routing tuples for join processing over multiple XML streams. The method is generic and can also be used for other data models.

To demonstrate the real-world applications of the RRPJ framework, a prototype for continuous and progressive processing of RSS feeds, called *Danaïdes* is presented in [42]. In *Danaïdes*, users pose queries in a SQL dialect. *Danaïdes* supports structured queries, spatial query and similarity queries. The *Danaïdes* service continuously processes the subscribed queries on the referenced RSS feeds and, in turn, published the query results as RSS feeds. Whenever memory is full, *Danaïdes* uses the RRPJ framework to determine the RSS feeds that are flushed to disk. The results of *Danaïdes* is a RSS feed, which can be read by standard RSS readers.

7.6 Progressive Approximate Joins

Users often do not require a complete answer to their query but rather only a sample. They expect the sample to be either the largest possible or the most representative (or both) given the resources available. We call the query processing techniques that deliver such results 'approximate'. Processing of queries to streams of data is said to be 'progressive' when it can continuously produce results as data arrives. In this chapter, we are interested in the progressive and approximate processing of queries to data streams when processing is limited to main memory. In particular, we study one of the main building blocks of such processing: the progressive approximate join. We devise and present several novel progressive approximate join algorithms. We empirically evaluate the performance of our algorithms and compare them with

algorithms based on existing techniques. In particular we study the trade-off between maximization of throughput and maximization of representativeness of the sample.

In data stream applications [3, 2, 11] the amount of data to be processed is generally much larger (potentially infinite) than the available memory. This is particularly true for applications whose processing is running on devices such as handheld computers, for instance. The progressive production of results therefore requires query processing algorithms that can make the best use of main memory and utilize secondary storage cleverly. Representative of this family of algorithms are the progressive joins such as XJoin [46], RPJ [40], HMJ [32], and our own RRPJ [44].

In addition, in many such applications, users are so concerned with rapid production of results that they are ready to give up completeness of the result. In this case, users may prefer results that can be produced in main memory only. In other words, users often do not require a complete answer to their query but rather only a sample. They expect the sample to be either the largest possible -they favor quantity-, the most representative -they favor quality- or both? They may need to seek a compromise between quality and quantity-, given the resources (main memory) available. We call the query processing techniques that deliver such results 'approximate'.

In this section, we discuss the design of progressive, approximate join algorithms. The reference progressive approximate join is *Prob* introduced in [14] and its extended version [15]. The authors propose the notion of maximum subset (MAX-Subset) which leads to similar strategies as the ones used by progressive algorithms such as RPJ [40] and RRPJ [44] to maximize the size of the set of results produced. The focus of the work is on result quantity. We show that the performance of *Prob* can be improved by stratifying the memory available. We propose *ProbHash*, a direct extension of *Prob*, in which the memory is hash partitioned and an approximate version of our progressive algorithm RRPJ also using hash partitioning. Interestingly, the authors of [14] have disqualified reservoir sampling based methods based on the extreme scenario given in [12] without further experiments. We show that this disqualification is mistaken. We propose a reservoir sampling-based approximate progressive join, that we call Reservoir Approximate Join (*RAJ*), and its stratified version *RAJHash*. We show that these algorithms favor the representativeness of the set of results produced and ensure better quality than the other algorithms.

7.6.1 Extreme Scenario

In this section, we describe a static and dynamic case for the extreme scenario shown in [12]. In the extreme scenario, the data distributions for the relations to be joined are skewed.

It was noted that when the data distributions are skewed, the join of the samples would not produce any results [12]. Given two relations $R_1(A,B)$ and $R_2(B,C)$, where A, B and C are attributes of the relations. Each relation consists of N tuples. Figure 7.3 shows the data in each of the relations. Suppose we obtain two random samples S_{R_1} and S_{R_2} from R_1 and R_2 respectively. The likelihood that the value b_1 is selected and included in sample S_{R_1} will be very low. Similarly, the likelihood that

the value b_2 is selected included in sample S_{R_2} is also very low. Thus, if we compute the samples first, and then compute $S_{R_1} \bowtie S_{R_2}$, the results will be empty. We refer to this as the static case. In the static case, we assume that all the data is available, and we first create the samples from each of the relations. The join is performed only when the samples are created.

A	B	B	C
a_1	b_1	b_2	c_1
a_2	b_2	b_1	c_2
a_3	b_2	b_1	c_3
a_4	b_2	b_1	c_4
...
a_N	b_2	b_1	c_N
R_1		R_2	

Fig. 7.3 Extreme Case.

In contrast, we consider the dynamic case for data streams applications. In the dynamic case, we progressively build the sample and perform the join at the same time. Assume that we maintain two samples S_{R_1} and S_{R_2} , each of size n . In this example, we set $n = 2$. Reservoir sampling [48] is used to maintain the two samples. When the tuple $R_1(a_1, b_1)$ arrives, we first probe S_{R_2} to find any tuples that can be joined. Since S_{R_2} is empty, no results are produced. We then insert $R_1(a_1, b_1)$ into S_{R_1} . Next, when the tuple $R_2(b_2, c_1)$ arrives, we probe S_{R_1} . Similarly, no results are produced. $R_2(b_2, c_1)$ is inserted into S_{R_2} . When the tuple $R_1(a_2, b_2)$ arrives, the probe of S_{R_2} will generate one result. It is then inserted into S_{R_1} . Similarly, when the tuple $R_2(b_1, c_2)$ arrives, it will join with the tuple in S_{R_1} . As the two samples are now full, when the next tuple arrives for R_1 , it will have a probability of $2/3$ to replace a randomly selected tuple in the reservoir. Since there are only two tuples in the sample, the probability that the rare tuple $R_1(a_1, b_1)$ is replaced is $1/3$. When the size of the reservoir is large, the probability that the rare tuple will be replaced in the dynamic case will be small. Thus, for the dynamic case, join results will still be produced even for skewed distributions.

7.6.2 Measuring the Performance of Progressive, Approximate Joins

There are two ways to measure the performance of an approximate algorithm. If we are interested in quantity, the measure of performance for the algorithm is the amount of results produced. If we are interested in quality, we need to measure the similarity between the data distribution of the complete set of results and the data distribution of the set of results produced. Because we are interested in progressive algorithms, performance is not a unique number but a function of time. It is measured in terms of throughput, quantity over time, when size matters. It is measured in terms of quality over time (quality throughput), when quality matters. If both

quantity and quality matter, we need both functions. Notice that the comparison of both functions by looking at quantity as a function of quality (or vice versa) at given points in time visualizes the compromise realized by a given algorithm.

We considered defining a combined measure of quantity and quality (similarly to the F-measure, which combines recall and precision). Unfortunately, our measure of quality using JS Divergence or any comparable statistical measures is unbounded, and cannot be normalized.

In order to measure quality, we need to compare two data distributions. We can compute, combine and compare any statistics and obtain more or less significant measurements at different level of granularity.

A reasonable metric is the Mean-Square Error (MSE) between the normalized histograms of the complete results and result produced by the approximate join. We, however choose a slightly more accurate measurement with the Jensen-Shannon divergence [28], which determines the similarity (or divergence) between two probability distributions.

We first discuss the MSE measure. The MSE measure measures the error differences between the actual and observed results produced. In the approximate join scenario, the actual results refer to the results produced if the entire join is computed (or when the memory is unlimited and all data fit into main memory). The observed results refer to the results produced by the approximate join method. In order to ensure a fair comparison between the actual and observed result distribution, we compare the normalized frequency instead of the actual frequency for each join attribute value. Let the total number of results produced by the complete and approximate join be $|R|$ and $|R'|$ respectively. For each value $v_i \in V$, where V denotes the domain of the join attribute, and $1 \leq i \leq |V|$, $|v_i|$ and $|v'_i|$ denotes the number of actual and observed results with value v_i . For each join attribute v_i , the normalized value for the complete and approximate joins is given by $\frac{|v_i|}{|R|}$ and $\frac{|v'_i|}{|R'|}$ respectively. The MSE between the complete join J and approximate join J' is given by

$$MSE(J, J') = \sum_{i=1}^V \left(\frac{|v_i|}{|R|} - \frac{|v'_i|}{|R'|} \right)^2 \quad (7.5)$$

In probability and information theory, the Kullback Leibler (KL) and Jensen-Shannon divergence are used to measure the similarity between two probability distributions, P and Q . We use the Jensen-Shannon divergence to measure the similarity between the actual (P) and observed result (Q) distribution. We measure the result quality produced by the approximate join using the Jensen-Shannon divergence. The Jensen-Shannon divergence measures the similarity between the actual result distribution (produced by a join where all tuples fit in memory) and the approximate join result distribution. Let $p(v_i) = \frac{|v_i|}{|R|}$ and $q(v_i) = \frac{|v'_i|}{|R'|}$. Before defining the Jensen-Shannon divergence, we first define the KL divergence, given as follows:

$$D_{KL}(P||Q) = \sum_{i=1}^V p(v_i) \log(p(v_i)/q(v_i)) \quad (7.6)$$

The Jensen-Shannon divergence is given by

$$D_{JS}(P||Q) = \frac{1}{2}D_{KL}(P||M) + \frac{1}{2}D_{KL}(Q||M) \quad (7.7)$$

where $M = \frac{1}{2}(P + Q)$

The goal is to minimize either the MSE or the JS divergence. When the value for either MSE or JS divergence is zero, the result distributions from the complete and approximate joins are exactly the same.

Given two approximate join methods, J_1 and J_2 , we say that J_1 produces better quality results than J_2 if the $QMeasure(J_1) < QMeasure(J_2)$. $QMeasure(Z)$ refers to either computing $MSE(Z)$ or $D_{JS}(Z)$. Z refers to an arbitrary approximate join method.

7.6.3 Different Types of Progressive, Approximate Joins

In this section, we describe five methods for performing approximate joins: (1) *Approximate RRPJ* (ARRPJ), (2) *Prob*, (3) *ProbHash*, (4) *Reservoir Approximate Join* (RAJ) and (5) *Stratified Reservoir Approximate Join* (RAJHash).

We first present the key idea for an existing progressive approximate join algorithm, *Prob*. Next, we propose the modification of an existing progressive join algorithm for approximate join processing, called *Approx-RRPJ*. Lastly, we propose three new algorithms (*ProbHash*, *RAJ* and *RAJHash*). *ProbHash* aims to maximize the result quantity as well as improve the overall throughput. Both *RAJ* and *RAJHash* are designed to optimize the result quality.

7.6.3.1 Approximate Join Framework

We first discuss a general framework for designing approximate join algorithms. The framework explores the tradeoffs between result quantity and quality.

Given two data streams $S_1(A,B)$ and $S_2(B,C)$. A , B and C are attributes of the data streams. Let the i -th tuple from S_1 and the j -th tuple from S_2 be denoted by $t_{S1}(a_i, b_i)$ and $t_{S2}(b_j, c_j)$ respectively. An approximate join is used to join the tuples from the two streams. The size of the memory available for query processing is small relative to the size of the data streams, which can be unbounded. When a new tuple arrives and memory is full, we will need to selectively discard some tuple(s) from memory. Indeed, an important design criteria for an effective approximate join algorithm is how tuples are discarded.

We first consider approximate join algorithms which maximizes the quantity of the results produced. We call such a algorithm $DP_X(k)$, which discards k tuples whenever memory is full. The goal of the $DP_X(k)$ policy is to maximize the expected size of the result subset. To achieve this, we can model the probability of the join attribute value(s) for tuples arriving on both streams. Let the arrival probabilities be $P_{S1}(B)$ and $P_{S2}(B)$ for streams S_1 and S_2 respectively. Whenever a tuple arrives, we

assign a priority to the tuple based on the arrival probabilities from the corresponding stream. For example, when a tuple $t_{S_1}(a_i, b_i)$ arrives, its priority value is given by $P_{S_2}(b_i)$. Similarly, the priority of a tuple $t_{S_2}(b_j, c_j)$ can be computed using $P_{S_1}(b_j)$. A possible implementation for $DP_X(k)$ is to maintain two priority queues (in ascending priority order) for the data streams. Whenever memory is full, $DP_X(k)$ discards the first k tuples taken from both streams. The intuition is that by keeping in memory tuples which have higher probability of joining with tuples from the other stream, the expected number of results produced will be maximized [40].

Next, we consider approximate join algorithms which are sampling-based. The goal is to optimize the quality of the results produced. We call such a algorithm DP_Y . DP_Y continuously maintains a random uniform sample for each of the data streams. When the memory is not full, tuples are inserted into the respective reservoirs. When memory is full, DP_Y determines whether the newly arrived tuple should be discarded, or be used to replace a tuple from the reservoir. Suppose the size of the memory is M , which is divided equally between the two streams S_1 and S_2 . Suppose n_{S_1} and n_{S_2} tuples have arrived for stream S_1 and S_2 respectively. We assume that the number of tuples that have arrived for each stream is greater than the available allocated memory (i.e., $n_{S_1} > (M/2)$, and $n_{S_2} > (M/2)$). A newly arrived tuple $t_{S_1}(a_i, b_i)$ has a $\frac{(M/2)}{n_{S_1}}$ chance of being used to replace a tuple in the reservoir. Similarly, for a tuple from S_2 . Even though DP_Y might not maximize the number of results produced, the quality of the results produced could be much better than $DP_X(k)$. This is because DP_Y ensures that the uniformity of the samples for each of the data streams. When a new tuple arrives, it is used to probe the corresponding reservoir. Mindful readers might note that DP_Y might not work well for skewed data streams if the memory is allocated equally between the two reservoirs. In this chapter, we show how we can tackle this problem by dynamically allocating memory for the reservoirs.

7.6.3.2 Approximate RRPJ (ARRPJ)

The Result-Rate Based Progressive Join (RRPJ) [44] is a progressive join algorithm. It builds statistics on the result distribution of the hash partitions. The goal of RRPJ is to maximize the number of results produced by using the result distribution statistics to determine the non-productive tuples to be flushed to disk whenever memory is full. In RRPJ, when all the tuples have arrived, a cleanup phase is invoked to compute the complete results for the join query.

In order to build a progressive, approximate join, we modify the design of RRPJ, such that only the in-memory processing phase is used during join processing. The clean-up phase which ensures the complete production of results is removed from the design. We call this join algorithm, *Approximate RRPJ (ARRPJ)*. Whenever memory is full, ARR PJ flushes tuples from memory. The tuples are discarded instead of being flushed to disk partitions.

7.6.3.3 Prob

The *PROB* [14, 15] approximate join is an instantiation of $DP_X(1)$. The goal of *PROB* is to maximize the quantity of results produced. It assigns a priority to each tuple that arrives. *Prob* can make use of either a fixed or variable memory allocation to store tuples from each of the data streams. For fixed allocation, two priority queues are used, one for each of the data streams. For variable allocation, a single priority queue is used for both streams. The priority for a tuple is determined by the arrival probabilities of the partner stream. We describe how *Prob* works. Given two streams S_1 and S_2 , a memory size M . Two priority queues, PQ_1 and PQ_2 , (one for each stream) are created. Using a fixed memory allocation, the size of each priority queue is $\frac{M}{2}$. In order to deliver results progressively, a probe-and-insert paradigm is used. When a tuple t_{S_1} arrives, it needs to probe all the tuples in the PQ_2 in order to determine join matches. Similarly, when a tuple t_{S_2} arrives, it needs to probe all the tuples in PQ_1 for join matches. At time τ , given that $|S_1|$ and $|S_2|$ tuples have arrived for S_1 and S_2 respectively. Using a variable memory allocation scheme, the size of the single priority queue is M . Whenever tuples arrive from either stream, it will have to scan all the tuples in the priority queue. The time complexity for both the fixed and variable memory allocation is given by $O(M(|S_1| + |S_2|))$.

7.6.3.4 ProbHash

In order to reduce the need to probe all in-memory tuples, we propose a progressive join algorithm, *ProbHash*. *ProbHash* relies on hash partitions to organize the in-memory tuples. In essence, *ProbHash* is a CPU-efficient extension of *Prob* [14, 15].

ProbHash organizes the in-memory tuples for each stream by storing the tuples using p priority queues, instead of a single priority queue. The value of p is dependent on the hash function used. The tuples in each priority queue are organized based on an ascending priority order. We denote the set of priority queue for data stream S_i as PQ_{S_i} ($1 \leq i \leq 2$). Figure 7.4 shows the two sets of priority queues. Whenever a tuple t_{S_1} arrives, its hash value is computed by the hash function (denoted by \oplus). It is then used to probe one of the priority queues in PQ_{S_2} . If join matches are found, the result is delivered to the user. After which, t_{S_1} is inserted to one of the priority queues of PQ_{S_1} . The set of priority queues, PQ_{S_1} and PQ_{S_2} , are each allocated $\frac{M}{2}$ memory. Within each priority queue set, we make use of a variable memory allocation scheme which allows the size of the priority queues to grow or shrink dynamically. This mitigates the effect of skewed data distribution, and ensure that the memory can be better utilized. Suppose the average length of each priority queue is L ($L \ll M$), the time complexity for *ProbHash* is given by $O(L(|S_1| + |S_2|))$.

When memory is full ($|S_1| + |S_2| = M$), and a new tuple arrives, we will need to select a tuple to be discarded from amongst the $2p$ priority queues. We first identify the priority queue PQ_i ($1 \leq i \leq 2p$) which contains the tuple with the smallest priority value. The complexity for finding the queue which contains a tuple with the smallest priority value is given by $O(p)$. This is because we only need to scan the

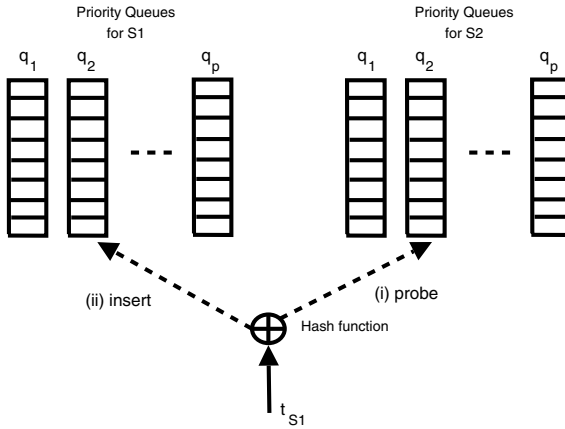


Fig. 7.4 Priority Queue for S_1 .

first element of each of the $2p$ priority queues. In the case of a tie (i.e. several queues with tuples having the smallest priority value), we randomly pick a tuple from one of these queues. Other methods can be used too (e.g., the tuple's age and preferring tuples that are older). We dequeue the tuple with lowest priority. We then compute the hash value for the newly arrived tuple, which is used to determine the priority queue it is inserted into. Due to the variable memory allocation, it is important to note that the sizes of the priority queues are not fixed. Hence, if the data distribution is skewed, some priority queues will be longer.

7.6.3.5 Reservoir Approximate Join (RAJ)

Conventional reservoir sampling [48] is used to produce a fixed size random sample of data. Algorithm 2 describes the details. While data is arriving (line 2), we get the next tuple from the data stream S (line 3). n denotes the total number of tuples that have arrived so far. If the number of tuples in the reservoir is less than the reservoir size $|R|$, we insert the tuple into the reservoir (line 5 to 6). Otherwise, the tuple is inserted into the reservoir with probability $\frac{|R|}{n}$ (line 8 to 10).

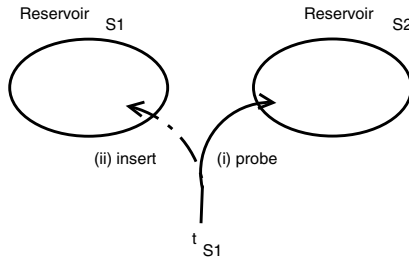
Conventional reservoir sampling can also be used in a progressive approximate join. We call this the Reservoir Approximate Join (RAJ). This is illustrated in Figure 7.5. Given two streams S_1 and S_2 , and memory with size M . Two reservoirs, $Reservoir_{S1}$ and $Reservoir_{S2}$ are created. Each reservoir is allocated $\frac{M}{2}$ memory. For each reservoir, the conventional reservoir sampling technique is used to manage the reservoir. When a tuple t_{S1} arrives, it is used to probe $Reservoir_{S2}$. Results (if any) are produced. Then, t_{S1} is inserted into $Reservoir_{S1}$. The problem with this approach is that the entire reservoir needs to be scanned in order to find tuples which can be joined with the newly arrived tuple.

Algorithm 2. Conventional Reservoir Sampling.

```

1:  $n = 0$ 
2: while ( !endOfStream(S) ) do
3:   Tuple  $t = \text{getNextTuple}(S)$ 
4:    $n = n + 1$ 
5:   if (  $n < |R|$  ) then
6:     Insert  $t$  into  $R$ 
7:   else
8:     Randomly generate a number  $\rho$  between 1 and  $n$ 
9:     if (  $\rho < |R|$  ) then
10:      Replace the  $\rho$ -th tuple in  $R$  with  $t$ 
11:     end if
12:   end if
13: end while

```

**Fig. 7.5** Reservoir Approximate Join.**7.6.3.6 Stratified Reservoirs Approximate Join (RAJHash)**

In statistics, stratified sampling [13] is another effective technique for sampling from a population. In stratified sampling, the population is divided into disjoint k sub-populations of sizes N_1, N_2, \dots, N_k respectively. Each sub-population is called a stratum, and is mutually exclusive (i.e., every element in the population must be assigned to only one stratum). Hashing is an effective way to assign each element to exactly one stratum. In order to reduce the need to scan the entire reservoir during probing, we adopt the idea of stratified sampling to organize the reservoir for each stream into multiple sub-reservoirs. In (*RAJHash*), a partitioning scheme is used to organize the tuples in each of the reservoir. The partitioning scheme effectively organizes the tuples into sub-population. In (*RAJHash*), this corresponds to the sub-reservoirs that are maintained. We call this algorithm the Stratified Reservoirs Approximate Join (*RAJHash*).

In the stratified reservoir approach, we allocate $\frac{M}{2}$ memory to each reservoir. Each reservoir consists of k sub-reservoirs. For each reservoir, a variable memory allocation scheme is used to allocate memory for the sub-reservoirs. Given a tuple t , the hash function, $f(t) = t.\text{value} \bmod k$, is used to assign the tuple to one of the

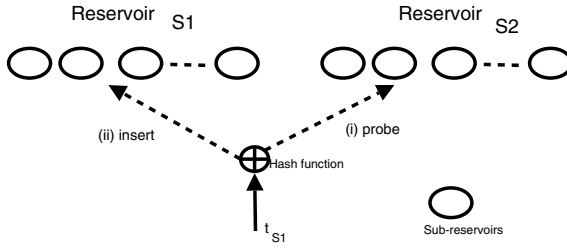


Fig. 7.6 Progressive Approximate Join using Stratified Reservoirs.

sub-reservoirs. t .value denotes the value of the join attribute. Algorithm 3 describes the insertion of a newly-arrived tuple using the stratified reservoir. In Line 1, h denotes the hashed value of the tuple. If n is less than $|R|$, then we will just add the tuple to the h -th sub-reservoir (Line 4). If n is greater or equal to $|R|$, then we will need to determine whether to replace a tuple from the reservoir with the newly arrived tuple (Line 6-10). To do this, a random number, ρ (between 1 and n) is generated. If ρ is greater than $|R|$, we discard t . Otherwise, t is used to replace a tuple from the h -th sub-reservoir. In this case, even though ρ is less than $|R|$, ρ can be greater than the size of the h -th sub-reservoir. To find the tuple to be replaced, we compute $i = \rho \bmod S$ (where S is the size of the h -th sub-reservoir). We then replace the i -th tuple in the h -th sub-reservoir.

As an implementation optimization, Algorithm 3 first chooses the sub-reservoir using the hash function, and then replaces a random tuple in the specific sub-reservoir. It is important to note that in order to maintain a simple random sample for each of the reservoirs, the decision on the tuple to be replaced should not be restricted to just a single reservoir. Instead, a random tuple from any of the sub-reservoirs can be replaced.

RAJHash introduces some advantages over *RAJ*. Firstly, it is more CPU-efficient as it reduces the number of in-memory tuples that are probed to identify join matches. Secondly, even in the presence of a skewed distribution, it is able to gracefully allocate more memory for sub-reservoirs which need a large sample, and less memory for sub-reservoirs which contain the skewed values. This is due to the variable memory allocation for the sub-reservoirs.

In this example, we illustrate how Stratified Reservoir works. Given two streams $S_1 = \{10, 22, 34, 11, 30, 90, 2, 1, 13, 10\}$ and $S_2 = \{10, 48, 20, 35, 12, 58, 67, 71, 44, 83\}$. In this example, the size of the memory $M = 10$ tuples. Two reservoirs $Reservoir_{S_1}$ and $Reservoir_{S_2}$ are created for S_1 and S_2 respectively. Each reservoir can hold 5 tuples. In addition, each reservoir is allocated 10 sub-reservoirs. The hash function $f(t) = t.value \bmod 10$ is used to allocate a tuple to one of the 10 sub-reservoirs. We denote a sub-reservoir for stream S_i as $reservoir_i^j$ ($0 \leq j < 9$) respectively.

For stream S_1 , the first tuple arrives. This is inserted into $reservoir_1^0$. Next, a tuple from S_2 arrives. This is first used to probe $Reservoir_{S_1}$, which in turn re-directs it to

Algorithm 3. Stratified Reservoir - Inserting a tuple.

```

1:  $h = f(t)$ 
2:  $n = n + 1$ 
3: if ( $n < |R|$ ) then
4:   Insert  $t$  into the  $h$ -th sub-reservoir
5: else
6:   Randomly generate a number  $\rho$ 
   between 1 and  $n$  (inclusive)
7:   if ( $\rho < |R|$ ) then
8:      $S =$  Get the size of the  $h$ -th sub-reservoir
9:      $i = \rho \bmod S$ 
10:    Replace the  $i$ -th tuple with  $t$ 
11:   else
12:     Discard  $t$ 
13:   end if
14: end if

```

sub-reservoir $reservoir_1^0$ which produces a result. After 5 tuples have arrived from each of the data streams, we have the following $reservoir_1^0 = \{10, 30\}$, $reservoir_1^1 = \{11\}$, $reservoir_1^2 = \{22\}$, $reservoir_1^4 = \{34\}$, $reservoir_2^0 = \{10, 20\}$, $reservoir_2^2 = \{12\}$, $reservoir_2^5 = \{35\}$ and $reservoir_2^8 = \{48\}$. When the sixth tuple from S_1 arrives, $Reservoir_{S_1}$ is full. We need to decide whether to discard a tuple from $Reservoir_{S_1}$. First, we compute the hash value of the sixth tuple to be 0 (i.e., $90 \bmod 10$). In order to determine whether to discard the tuple, we randomly generate a number ρ between one and six (inclusive). If $\rho \leq 5$, then we will replace a tuple in the sub-reservoir $reservoir_1^0$ with this newly arrived tuple. Suppose the value of ρ is 4. It is important to note that there are only two tuples in $reservoir_1^0$. To determine which tuple to be replaced, we compute $4 \bmod 2 = 0$. Thus, the first tuple (value=10) is then replaced with the newly arrived tuple. Thus, sub-reservoir $reservoir_1^0 = \{90, 30\}$. Similarly, when the sixth tuple (value = 58) from S_2 arrives, we need to decide whether to discard or replace a tuple from $reservoir_2^8$. We generate a random number, ρ between one and six (inclusive). Suppose $\rho = 6$. Thus, we discard the newly arrive tuple. Thus, sub-reservoir $reservoir_2^8 = \{48\}$.

7.6.4 Discussion

Approx-RRPJ, *Prob* and *ProbHash* attempt to maximize the quantity (i.e., number of results produced) by sacrificing tuples that do not produce any or produce few results. Therefore, they tend to favour results in certain ranges. In contrast, *RAJ* and *RAJHash* strive to maintain a good representative sample. With limited memory, an approximate join algorithm need to effectively make use of the available memory, balancing between quantity and quality of the results produced.

7.7 Open Issues

In this section, we discuss several open research issues. In the generic progressive join framework, one of the key factors that contribute to the efficiency and effectiveness of the framework is an effective partitioning method. A good partitioning method provides a uniform distribution of the data into multiple partitions. It reduces the number of tuples that need to be probed during join processing. In the chapter, we have studied the use of hash partitions for relational and XML data, two-dimension grid for spatial data, and multi-dimension grid for high-dimensional data. However, for both existing and new data models, an open issue lies in finding an effective partitioning scheme. In data stream processing, the data distribution can vary over time. While an effective partitioning scheme can impose a uniform distribution of data into multiple partitions for the initial data, it may not be effective for future data that is of a different data distribution. An open issue that needs to be solved is the design of an adaptive partitioning function that can adapt to evolving data.

The work on progressive, approximate joins showed that the use of sampling is an attractive primitive. The framework focuses on balancing between quality and quantity, and allows it to be easily generalized for other data models (e.g spatial, high-dimensional, XML). In this chapter, we have discussed two families of progressive approximate join algorithms which either maximize the quantity or quality of the results produced. *Prob* and *ProbHash* cannot be easily generalized to other data models. This is due to the dependence on the arrival probabilities of the partner data stream. While the arrival probabilities for relational data can be computed in a straightforward manner, it is difficult to compute such probabilities for data from other data models.

Another limitation of *Prob* and *ProbHash* is that they cannot be easily extended for multi-way approximate join, unless the multi-way join query plan is decomposed into a series of binary joins. This is because for a multi-way join, it is not clear which is the partner stream. Decomposing the multi-way join query plan to a series of binary joins would limit the adaptiveness of the join. One of the advantages of using *RAJ* and *RAJHash* is that they can be easily generalized to other data models. This is because the decision to discard a tuple from the reservoir (or sub-reservoirs) does not depend on the data model. For multi-way joins, multiple reservoirs can be defined for each of the data streams.

In order to address the tradeoff between the two families of algorithms, a tunable sampling approach can be used. The motivation for tunable sampling is to allow progressive approximate joins to balance between the quantity and quality of results produced. Tunable sampling is defined as a sampling technique which allows users to tune the type of sample produced by the sampling process. The sample can either favor the frequencies for including popular data values in the sample (i.e., quantity), or favor representativeness of the data (i.e., quality). Let C denote the set of criteria that the user wishes to maximize, and c_i denotes the individual criterion to be tuned ($c_i \in C, 1 \leq i \leq |C|$). Let W denote the set of weights assigned to each criterion, and

w_i denotes the individual weight assigned to criterion i . $\sum_{i=1}^{|C|} w_i = 1$. We consider $C = \{\text{Quantity, Quality}\}$. Next, we introduce the notion of inclusion probability. $P(t)$ is the probability that a tuple t will be included in the sample. We refer to this as the inclusion probability. Given a criterion c_i , the inclusion probability is given by $P_{c_i}(t)$. We formally define tunable sampling as follows: Given a set of criterias C , a set of criteria weights W , and the inclusion probability for each of the criterias. The combined inclusion probability for all the criteria is given by:

$$\sum_{i=1}^{|C|} w_i P_{c_i}(t) \quad (7.8)$$

For quantity maximization techniques (e.g., *Prob*, *ProbHash*), $P_{\text{Quantity}} = n_v / N$, where n_v denotes the number of tuples with value v , and N denotes the total number of tuples that have arrived so far. For quality maximization techniques (e.g., *RAJ*, *RAJHash*), $P_{\text{Quality}} = |R| / N$, where $|R|$ denotes the size of the reservoir, and N denotes the total number of tuples that have arrived so far.

In this chapter, we focus on the design of join algorithms. Besides join operators, query plans that are used in data streaming systems make use of different operators. An open issue is the holistic consideration of the entire query plan w.r.t to optimization for result delivery, instead of focusing only on join processing.

7.8 Conclusion

The universe of network-accessible data is expanding. Data streaming applications need to process data streams from heterogeneous remote sources. In these applications, the amount of memory available is limited either because of the intrinsic constraints of the processing device or because of the large data volume. Hence, it is important that the memory is effectively used during result production.

In this chapter we discuss the design and implementation of progressive and approximate join algorithms for data streams. The design is guided by several key requirements. The algorithm must be non-blocking (or progressive), i.e., it must be able to produce results as soon as possible. The algorithm must maximize either the result quantity or quality or ensure a compromise between both. We discuss the design of a novel generic progressive join framework, called Result-Rate based Progressive Join (RRPJ) framework. We present its various instantiations for different data models. Based on the foundation built for progressive join, we introduce several progressive and approximate join algorithms. Lastly, we discuss several open research issues that need to be solved in order to improve the state-of-art in progressive query processing.

References

1. Arge, L.A., Procopiuc, O., Ramaswamy, S., Suel, T., Vitter, J.S.: Scalable Sweeping-Based Spatial Join. In: VLDB, pp. 570–581 (1998)
2. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and Issues in Data Stream Systems. In: PODS, pp. 1–16 (2002)
3. Babu, S., Widom, J.: Continuous Queries over Data Streams. SIGMOD Record 30(3), 109–120 (2001)
4. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In: SIGMOD, pp. 322–331 (1990)
5. Berchtold, S., Keim, D.A., Kriegel, H.P.: The X-tree: An Index Structure for High-Dimensional Data. In: VLDB, pp. 28–39 (1996)
6. Böhm, C., Braunmüller, B., Breunig, M.M., Kriegel, H.P.: High Performance Clustering Based on the Similarity Join. In: CIKM, pp. 298–305 (2000)
7. Böhm, C., Braunmüller, B., Krebs, F., Kriegel, H.P.: Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. In: SIGMOD, pp. 379–388 (2001)
8. Böhm, C., Krebs, F.: Supporting KDD Applications by the k -Nearest Neighbor Join. In: Mařík, V., Štěpánková, O., Retschitzegger, W. (eds.) DEXA 2003. LNCS, vol. 2736, pp. 504–516. Springer, Heidelberg (2003)
9. Böhm, C., Krebs, F.: The k -Nearest Neighbour Join: Turbo Charging the KDD Process. Knowl. Inf. Syst. 6(6), 728–749 (2004)
10. Brinkhoff, T., Kriegel, H.P., Seeger, B.: Efficient Processing of Spatial Joins Using R-Trees. In: SIGMOD, pp. 237–246 (1993)
11. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring Streams - A New Class of Data Management Applications. In: VLDB, pp. 215–226 (2002)
12. Chaudhuri, S., Motwani, R., Narasayya, V.R.: On Random Sampling over Joins. In: SIGMOD, pp. 263–274 (1999)
13. Cochran, W.G.: Sampling Techniques, 3rd edn. John Wiley (1977)
14. Das, A., Gehrke, J., Riedewald, M.: Approximate Join Processing Over Data Streams. In: SIGMOD, pp. 40–51 (2003)
15. Das, A., Gehrke, J., Riedewald, M.: Semantic Approximation of Data Stream Joins. IEEE Trans. Knowl. Data Eng. 17(1), 44–59 (2005)
16. Dittrich, J.P., Seeger, B., Taylor, D.S., Widmayer, P.: Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In: VLDB, pp. 299–310 (2002)
17. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: SIGMOD, pp. 47–57 (1984)
18. Hellerstein, J.M., Avnur, R., Chou, A., Hidber, C., Olston, C., Raman, V., Roth, T., Haas, P.J.: Interactive data Analysis: The Control Project. IEEE Computer 32(8), 51–59 (1999)
19. Hong, M., Demers, A., Gehrke, J., Koch, C., Riedewald, M., White, W.: Massively Multi-Query Join Processing in Publish/Subscribe Systems. In: SIGMOD. ACM Press, Beijing (2007)
20. Huang, Y.W., Jing, N., Rundensteiner, E.: Spatial Joins using R-trees: Breadth-first Traversal with Global Optimizations. In: VLDB, pp. 396–405 (1997)
21. Ibrahim, I.K.: Handbook of Research on Mobile Multimedia (N/A). IGI Publishing, Hershey (2006)
22. Kalashnikov, D.V., Prabhakar, S.: Fast Similarity Join for Multi-Dimensional Data. Inf. Syst. 32(1), 160–177 (2007)

23. Koudas, N., Sevcik, K.C.: Size Separation Spatial Join. In: SIGMOD, pp. 324–335 (1997)
24. Koudas, N., Sevcik, K.C.: High Dimensional Similarity Joins: Algorithms and Performance Evaluation. In: ICDE, pp. 466–475 (1998)
25. Koudas, N., Sevcik, K.C.: High Dimensional Similarity Joins: Algorithms and Performance Evaluation. *IEEE Transactions on Knowledge and Data Engineering* 12(1), 3–18 (2000)
26. Lawrence, R.: Early Hash Join: A Configurable Algorithm for the Efficient and Early Production of Join Results. In: VLDB, pp. 841–852 (2005)
27. Li, F., Chang, C., Kollios, G., Bestavros, A.: Characterizing and Exploiting Reference Locality in Data Stream Applications. In: ICDE, p. 81 (2006)
28. Lin, J.: Divergence Measures based on the Shannon Entropy. *IEEE Transactions on Information Theory* 37(1), 145–151 (1991)
29. Lo, M.L., Ravishankar, C.V.: Spatial Joins Using Seeded Trees. In: SIGMOD, pp. 209–220 (1994)
30. Lo, M.L., Ravishankar, C.V.: Spatial Hash-Joins. In: SIGMOD, pp. 247–258 (1996)
31. Mamoulis, N., Papadias, D.: Integration of Spatial Join Algorithms for Joining Multiple Inputs. In: SIGMOD, pp. 1–12 (1999)
32. Mokbel, M.F., Lu, M., Aref, W.G.: Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In: ICDE, pp. 251–263 (2004)
33. Nelson, R.C., Samet, H.: A Population Analysis for Hierarchical Data Structures. In: Dayal, U., Traiger, I.L. (eds.) SIGMOD, pp. 270–277. ACM Press, New York (1987)
34. Patel, J.M., DeWitt, D.J.: Partition Based Spatial-Merge Join. In: SIGMOD, pp. 259–270 (1996)
35. Sellis, T., Roussopoulos, N., Faloutsos, C.: R+-tree: A Dynamic Index for Multi-Dimensional Objects. In: VLDB (1987)
36. Sevcik, K.C., Koudas, N.: Filter Trees for Managing Spatial Data over a Range of Size Granularities. In: VLDB, pp. 16–27 (1996)
37. Shim, K., Srikant, R., Agrawal, R.: High-Dimensional Similarity Joins. In: ICDE, pp. 301–311 (1997)
38. Srivastava, U., Widom, J.: Memory-Limited Execution of Windowed Stream Joins. In: VLDB, pp. 324–335 (2004)
39. Stark, M., Fernández, M., Michiels, P., Siméon, J.: XQuery streaming á la Carte. In: ICDE (2007)
40. Tao, Y., Yiu, M.L., Papadias, D., Hadjieleftheriou, M., Mamoulis, N.: RPJ: Producing Fast Join Results on Streams through Rate-based Optimization. In: SIGMOD, pp. 371–382 (2005)
41. Tok, W.H., Bressan, S., Lee, M.L.: Progressive Spatial Joins. In: SSDBM, pp. 353–358 (2006)
42. Tok, W.H., Bressan, S., Lee, M.-L.: Danaïdes: Continuous and Progressive Complex Queries on RSS Feeds. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 1115–1118. Springer, Heidelberg (2007)
43. Tok, W.H., Bressan, S., Lee, M.-L.: Progressive High-Dimensional Similarity Join. In: Wagner, R., Revell, N., Pernul, G. (eds.) DEXA 2007. LNCS, vol. 4653, pp. 233–242. Springer, Heidelberg (2007)
44. Tok, W.H., Bressan, S., Lee, M.-L.: RRPJ: Result-Rate Based Progressive Relational Join. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 43–54. Springer, Heidelberg (2007)

45. Tok, W.H., Bressan, S., Lee, M.-L.: Twig'n Join: Progressive Query Processing of Multiple XML Streams. In: Haritsa, J.R., Kotagiri, R., Pudi, V. (eds.) DASFAA 2008. LNCS, vol. 4947, pp. 546–553. Springer, Heidelberg (2008)
46. Urhan, T., Franklin, M.J.: XJoin: Getting Fast Answers From Slow and Bursty Networks. Tech. Rep. CS-TR-3994, University of Maryland (1999), <http://citeseer.nj.nec.com/urhan99xjoin.html>
47. Urhan, T., Franklin, M.J., Amsaleg, L.: Cost Based Query Scrambling for Initial Delays. In: Haas, L.M., Tiwary, A. (eds.) SIGMOD, pp. 130–141. ACM Press (1998)
48. Vitter, J.S.: Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11(1), 37–57 (1985)
49. Wilschut, A.N., Apers, P.M.G.: Dataflow Query Execution in a Parallel Main-Memory Environment. In: PDIS, pp. 68–77 (1991)
50. Xia, C., Lu, H., Ooi, B.C., Hu, J.: Gorder: An Efficient Method for KNN Join Processing. In: VLDB, pp. 756–767 (2004)
51. Xie, J., Yang, J., Chen, Y.: On Joining and Caching Stochastic Streams. In: SIGMOD, pp. 359–370 (2005)

Chapter 8

Online Aggregation

Sai Wu, Beng Chin Ooi, and Kian-Lee Tan

Abstract. In this chapter, we introduce a new promising technique for query processing, online aggregation. Online aggregation is proposed based on the assumption that for some applications, the precise results are not always required. Instead, the approximate results can provide a good enough estimation. Compared to the precise results, computing the approximate ones are more cost effective, especially for large-scale datasets. To generate the approximate result, online aggregation retrieves samples continuously from the database. The samples are streamed to the query engine for processing the query. The accuracy of the approximate result is described by a statistical model. Normally, the result is refined as more samples are obtained. The user can terminate the processing at any time, when he/she is satisfied with the quality of the result.

The performance of online aggregation relies on the sampling approach and estimation model. In this chapter, our discussion is focused on these two components. Besides introducing the basic principles of online aggregation, we also review some new applications built on top of it. We complete the chapter by discussing the challenges of online aggregation and some future directions.

8.1 Introduction

In data warehouse systems, aggregation query is employed to create a statistical result for decision making. It is one of the most expensive queries in the database systems. As an example, some aggregate queries in real applications, such as the template queries in TPC-H [1] (a database benchmark for evaluation the performance of online analysis), involve multiple tables and a series of *joins* and *groupbys*. Processing such queries may last for hours. Moreover, as the size of

Sai Wu · Beng Chin Ooi · Kian-Lee Tan
School of Computing, National University of Singapore
e-mail: [wusai, ooibc, tankl}@comp.nus.edu.sg](mailto:{wusai, ooibc, tankl}@comp.nus.edu.sg)

processed data increases exponentially,¹ aggregation queries are becoming more costly. For example, Figure 8.1 shows the top 6 results for 10T TPC-H datasets. *QphH* denotes the number of processed queries per hour. As we can see, even the most costly servers cannot provide a satisfactory throughput for 10T TPC-H data.





10,000 GB Results					
Rank	Company	System	QphH	Price/QphH	Watts/KQphH
1		HP Integrity Superdome/Dual-Core Itanium/1.6 GHz	208,457	27.97 USD	NR
2		HP Integrity Superdome-DC Itanium2/1.6GHz/64p/128c	171,380	32.91 USD	NR
3	ORACLE	Sun Fire[™] E25K server	108,099	53.80 USD	NR
4	UNISYS	Unisys ES7000 Model 7600R Enterprise Server(16s)	80,172	18.95 USD	NR
5		HP Integrity Superdome	63,650	38.54 USD	NR
6		HP Integrity Superdome - Itanium2/1.5 GHz-64p/64c	49,104	118.13 USD	NR

Fig. 8.1 Top 6 Results for 10T TPC-H Dataset.

To address the scalable problem, new computing models, such as parallel databases and cloud systems, are being developed. But switching to the new computing model needs to change the basic infrastructure, which is not accepted by many companies. Fortunately, an observation of the real systems motivates a new processing technique, the approximate query processing (ApQP). In many decision making applications, approximate results can provide a good enough prediction. For example, suppose a retailer's monthly sales is \$350,123. If the prediction program returns \$348,000 as the result with confidence 95%, such estimation is good enough to provide a general view for the sales. Compared to other processing approaches, ApQP incurs less costs, as

1. ApQP does not need to scan the whole dataset to process a query. Some ApQP approaches sample the dataset to compute an approximate result, while others maintain a precomputed synopsis to speed up the query processing.
2. Users can indirectly control the cost of ApQP by specifying the required accuracy of the results. In most cases, ApQP incurs more overhead to generate a higher quality result.

ApQP approaches can be classified into two types based on how the approximate results are generated. [3] and [17] maintain samples or previous query results in the precomputed synopsis, which are used to process future queries. The precomputed synopsis can greatly reduce the processing cost, if the query can be fully or partially

¹ <http://www.b-eye-network.com/view/7188>

processed via the synopsis. However, the user cannot control the accuracy of the results, as samples in the synopsis are precomputed. Moreover, queries whose synopsis are maintained can take advantage of such a scheme. On the contrary, in [10] and [9], an online aggregation approach is adopted, where samples are continuously drawn from the database and the results are refined as more samples are retrieved. The advantage of online aggregation is that the user can control the processing of the queries. The approximate results are updated periodically and if the user is satisfied with current results, he can terminate the processing to save the cost. If the user does not specify his preference, the query will run to the end and the precise results are returned. In comparison, some other ApQP approach cannot provide precise results, which limits their usage.

The processing strategy of online aggregation is similar to the one used in the continuous queries on data streams (see Chapter 7 for details). Continuous samples retrieved from the database can be considered as a stream and the query is evaluated against the stream. This observation motivates the work in [21], where a sample stream is generated to feed multiple queries. Compared to the conventional stream processing, in online aggregation, the processing engine does not maintain a buffer for processed samples. Instead, statistics of data distribution are updated to measure how good the approximate result is.

The basic principle of the online aggregation is derived from well-known statistics theorems. It can be easily embedded into existing database systems, such as Postgres. Although the idea of online aggregation was proposed more than 10 years ago, it attracts increasing research efforts in recent years, due to the requirement of handling large-scale data analysis job. Such job normally involves terabytes of data and the cost of computing accurate results is prohibitively high. Therefore, online aggregation techniques are applied to effectively reduce the processing cost. To address the data scale problem, different enhancements of online aggregation are proposed:

- **Online Aggregation for Multiple Relations**

The basic online aggregation technique is designed for computing the aggregation results on one single relation. In data warehouse systems, user queries always involve multiple relations. Therefore, different schemes [9] [15] [12] are proposed to extend the technique to handle aggregations on multiple relations efficiently.

- **Online Aggregation for Multiple Queries**

Many systems are designed to support multiple concurrent users. By sharing query results among the users, the system can effectively reduce the cost, as the same query does not need to be processed repeatedly. The idea of sharing samples/results can be applied to the online aggregation processing as well [21]. Queries are organized into a dissemination graph, where samples/results are shared between the neighbor nodes.

- **Distributed Online Aggregation**

Centralized system is not capable of supporting extremely large data and new distributed and parallel database systems are implemented to address the problem. To support online aggregation in distributed databases, the sampling approach needs to be redesigned and the computation needs to be distributed to

multiple nodes [20]. The performance is improved significantly as both sampling and query processing are handled in parallel.

- **Online Aggregation in MapReduce**

MapReduce [8] is a new parallel processing framework proposed by Google. It is widely used in decision making system, scientific computation system and data warehouse system. MapReduce can be enhanced with online aggregation techniques [6] to further improve its performance.

In this chapter, we will review current efforts in adopting online aggregation techniques in database systems. The rest of the chapter is organized as follows. Section 8.2 introduces the basic principle of the online aggregation. In particular, we present the statistical model and sampling approaches. In Section 8.3, we give an overview of above new applications that are implemented by extending the basic online aggregation model. And in Section 8.4, we conclude the chapter and discuss the future and challenges of this new processing technique.

8.2 Basic Principles

Online aggregation is first proposed in [10]. It is based on the assumption that aggregate queries are typically used to get a “rough picture” from a large amount of data and thus, instead of producing a precise answer, an “approximately correct” answer suffices. The basic idea is to generate approximate results with randomly selected samples and refine the results as more samples are retrieved. To indicate how good the result is, a confidence c and an error bound ε are provided with each result. Suppose the precise result is v and the approximate one is \bar{v} , we have $P(|v - \bar{v}| \leq \varepsilon) \geq c$. Namely, the probability of the difference between the real result and estimated result being less than ε is c .

In this chapter, we use typical aggregation queries (average, sum and count) as the examples. Consider a typical single relational aggregate query such as

```
SELECT op(expression( $x_i$ )) FROM  $T$ 
```

To process the query, k random samples are retrieved from relation T . Let S denote the sample set. The approximate result \bar{v} is computed as follows:

1. $\bar{v} = \frac{|T|}{k} \sum_{\forall t_i \in S} c(\text{expression}(t_i))$, where $|T|$ is the size of table T , $c(x) = 1$ if $op = \text{count}$ and $c(x) = x$ if $op = \text{sum}$.
2. $\bar{v} = \frac{1}{k} \sum_{\forall t_i \in S} \text{expression}(t_i)$, when $op = \text{avg}$.

As an example, Table 8.1 lists the scores of students in class 1. The average score is 69.625 and the total score is 557. If we pick random samples of students 1001, 1003 and 1008, the estimated average score and total score are $68.67 = \frac{63+82+61}{3}$ (with error rate $1.37\% = \frac{|69.625-68.67|}{69.625}$) and $549.33 = \frac{(63+82+61) \times 8}{3}$ (same error rate as the average score), respectively. As more samples are retrieved from the table, we can get a more precise result. However, the sampling cost increases as well.

Table 8.1 Student Record.

class ID	student ID	score
1	1001	63
1	1002	65
1	1003	82
1	1004	58
1	1005	66
1	1006	73
1	1007	89
1	1008	61

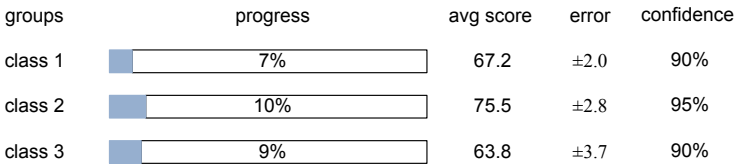


Fig. 8.2 Scenario of Online Aggregation.

Figure 8.2 illustrates the interface of an online aggregation system. Suppose we have a table for the students in all classes and try to calculate the average score of each class. Online aggregation returns an approximate result for each class with the corresponding estimated error bound and confidence. The sampling progress is provided to indicate how many samples have been processed. In Figure 8.2, the average score of class 1 is about 67.2 ± 2.0 with confidence 90%. Currently, 7% tuples are used as the samples to compute the result. If the user is satisfied with the result, he can stop the processing for class 1. And then no more samples are retrieved for class 1.

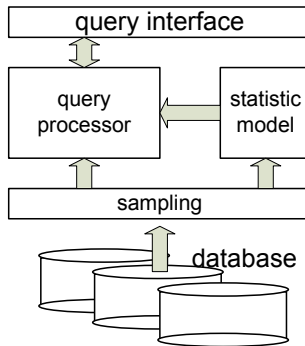


Fig. 8.3 Architecture of Online Aggregation.

Adopting online aggregation technique incurs small changes to current DataBase Management Systems (DBMSs). It is less intrusive and can be built on top of most existing systems. Figure 8.3 gives an overview of an online aggregation system. Specifically, given a query, a stream of random samples is generated from the local database. The samples are then forwarded to the query processor to generate an approximate result. The samples are also used in statistical analysis, where the central limit theorem is applied to estimate the error bound and confidence. The sample stream stops if the user terminates the query processing or all tuples are retrieved. The efficiency and effectiveness of the online aggregation relies on the statistical model and the sampling approach.

8.2.1 Statistical Model

The statistical model is employed to estimate the confidence and error bound of the approximate result. In [10], three types of running confidences are computed, namely conservative confidence, large-sample confidence and deterministic confidence. In this chapter, we focus on large-sample confidence, which is generated based on the central limit theorem and provides a good enough result without too much samples.

Given a random sample set S from table T with size k ($|S| = k$), each sample x_i in S can be used to compute an approximate result. Let function f denote the aggregation function. We have a set of independent variables $f(x_0), f(x_1), \dots, f(x_{k-1})$ ($x_i \in S$), which follow the same distribution. Suppose $f(x_i)$ has expectation μ and variance σ^2 . The central limit theorem states that the distribution of the sample average of these random variables approaches the normal distribution with a mean μ and variance σ^2 .

By estimating μ , we can generate the approximate result. However, the problem is, we have another variable σ , which needs to be estimated as well. In online aggregation, we compute the variance $\bar{\sigma}$ of the samples and apply it to simulate σ . As $Var(x) = E[x^2] - E[x]^2$, we have

$$\bar{\sigma}^2 = \begin{cases} \sum f(x_i)^2 - (\sum f(x_i))^2 & \text{if } op = avg \\ |T|^2 \sum f(x_i)^2 - \frac{|T|^2 (\sum f(x_i))^2}{k^2} & \text{if } op = sum \text{ or } op = count \end{cases}$$

Now, we use $\bar{\sigma}$ to replace σ . Let \bar{v} and v denote the approximate result and the real result, respectively. And we use ε to represent the error bound. Based on the central limit theorem,

$$P(\bar{v} - v) \leq \varepsilon \approx 2\phi\left(\frac{\varepsilon\sqrt{k}}{\sigma}\right) - 1 \quad (8.1)$$

Equation 8.1 defines the relationship between the error bound ε and sample size k . And the probability $P(\bar{v} - v) \leq \varepsilon$ is our confidence for the result.

As we have two variables in Formula 8.1 ε and P , we cannot solve the formula completely. A normal approach is that we set the value of one variable and compute

the other one. For example, if the expected confidence is 95%, then the formula is transformed into

$$\phi\left(\frac{\varepsilon\sqrt{k}}{\sigma}\right) = \frac{0.95 + 1}{2} \quad (8.2)$$

Based on current sample set, we can estimate the error bound ε correspondingly.

In online aggregation, a more commonly used approach is to refine the two variables iteratively. We can first set confidence to c_0 and then we get an error bound ε_0 . As more samples are retrieved, we set the error bound to ε_0 and compute the new confidence c_1 . This process continues and hence, both the confidence and error bounds are improved sequentially. When user is satisfied with the result, he can terminate the processing.

In this section, we discuss the most simple case, aggregations on a single relation. To handle complex queries, the basic statistical model has to be extended and we will discuss the details in Section [8.3](#).

8.2.2 Sampling

Previous statistical model is based on the assumption that all samples are randomly retrieved. Getting random samples seems to be easy, but in fact, it is a very challenging problem. In MySQL, the following SQL statement can be used to retrieve k random samples,

```
select * from T order by RAND() limit k
```

However, for a large dataset, the above query is very inefficient, as it involves scanning the whole table.

Sampling techniques have been well studied in the database community [\[16\]](#). To handle skewed data distribution, a new sampling technique, outlier-indexing, is proposed in [\[5\]](#). By combining weighted samples from uniform sampling and outlier-indexing, the hybrid scheme [\[5\]](#) can provide an aggregate result with significantly reduced approximation error. Sampling process may be too costly for large-size of samples. In [\[13\]](#), an algorithm is proposed to maintain large number of samples in disk and apply the precomputed samples to answer queries. The algorithm is suitable for both biased and unequal probability sampling. In [\[2, 4\]](#), samples are retrieved and stored in synopsis. New queries can exploit the precomputed samples to generate approximate result.

Different from traditional sampling approach, in online aggregation, the sampling process must guarantee the following properties.

- The samples should be randomly distributed over the dataset. Otherwise, the statistical model cannot generate a correct confidence and error bound.
- The samples cannot be repeated, because when the user does not terminate the processing, online aggregation should return a precise result.
- The sampling process should not be a performance bottleneck.

In [10], three sampling approaches are proposed for online aggregation, heap scan, index scan and sampling from indices.

In database, heap files are used as the basic storage file structure. The order of a heap file relies on the sequence of data insertion or some explicit clustering. If such order is not correlated with the attributes of the records, we can apply the heap scan to generate the random samples.

Generally speaking, index scan is not applicable to online aggregation, as index sorts the values (B-tree index) or groups the values (hash index) by some attribute. However, if the aggregated attribute is not correlated with the indexed attribute, we can still apply the index scan to generate usable samples.

In [16], a pseudo-random sampling for various index structures is proposed. The index sampling approach can provide a meaningful confidence interval. However, its performance is worse than the heap scan and index scan, because it incurs random I/Os for the indices.

Besides the three approaches, a scrambling approach [21] is introduced for providing continuous sampling streams for data warehouse systems, where data are seldom updated or the batch updates are applied. The scrambling approach permutes the records in a table multiple times, until the sequence of the records is not correlated to any explicit order. Hence, scanning the table will generate a random sample stream.

8.3 Advanced Applications

In the last section, we give the basic model of online aggregation. However, it is quite challenging to apply such a technique to real systems. In this section, we discuss some efforts that extend the technique to handle different problems. We focus on non-nested queries. For a discussion on nested queries, readers are referred to [19].

8.3.1 Online Aggregation for Multi-relation Query Processing

Previous discussion focuses on the single table case, where all samples are retrieved from the same table. But in practice, aggregations are often processed after joins. Therefore, we must extend the online aggregation model to handle multi-relation query. Consider the following query,

```
SELECT op(expression(xi)) FROM  $T_1, \dots, T_k$  WHERE predicate
```

To process the query, we can retrieve samples from the involved tables and perform the join for the samples. However, given random samples t_1, \dots, t_k from T_1, \dots, T_k respectively, with low probability, the samples can join together to generate a valid result. Therefore, online aggregation is inefficient due to lack of valid join results.

Only after a large portion of the tables has been scanned will we get a good enough aggregate result.

To address this problem, a special technique, ripple join [9], is proposed. Ripple join is similar to the nested loop join, which is also non-blocking. The intuition of ripple join is to retrieve samples from the tables iteratively. And once a sample is generated, it is joined with existing samples of other tables. To simplify the discussion, we focus on the two-table case $T_1 \bowtie T_2$; the same approach can be easily generalized to other cases.

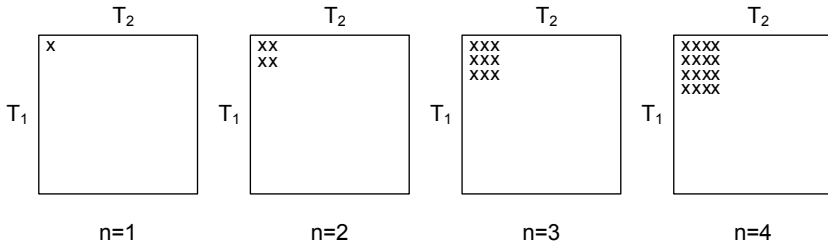


Fig. 8.4 Square Ripple Join.

Figure 8.4 illustrates the idea of ripple join. We use a matrix to denote the progress of sampling in each table. Each “x” in the matrix denotes a seen result or a block of seen results of the cartesian product $T_1 \times T_2$. In Figure 8.4, we retrieve samples from T_1 and T_2 in an even rate. Namely, after k steps, we have k samples from T_1 and T_2 , respectively. And we can generate k^2 results for the cartesian product, some of which are valid results and can be used to estimate the aggregate answer. In ripple join, once we get a sample from a table, we can join it with samples of the other table. For that purpose, all retrieved samples should be buffered in memory. Therefore, one limitation of the ripple join is the memory size. When the memory cannot hold all the retrieved samples, ripple join will switch to the normal nested loop join.

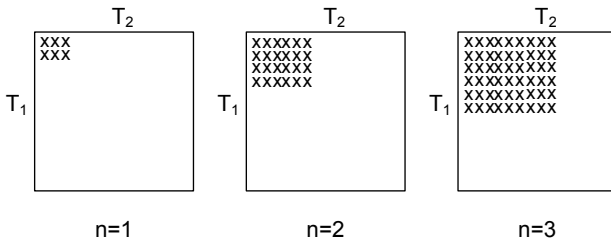


Fig. 8.5 Rectangle Ripple Join.

One advantage of ripple join is that we can adaptively adjust the sampling process. We can retrieve more samples from a table than the other to provide better confidence interval. As an example, suppose each T_2 tuple can join with 3 T_1 tuples, while each T_1 tuple can join with 2 T_2 tuples. We can set the sampling rates of T_1 and T_2 as 2:3 to generate more valid join result. Figure 8.5 illustrates the idea. In ripple join processing, we can even adaptively change the sampling rate of involved tables, based on the observed data distribution.

In ripple join, the central limit theorem is used to estimate the confidence, as discussed in the previous section. However, when multiple tables are involved, the model needs to be modified. Let $x_i \in T_1$ and $y_j \in T_2$, and (x_i, y_j) be a sample for $T_1 \bowtie T_2$. We make a similar assumption as before. Namely, after enough samples are retrieved, we assume the estimator follows the normal distribution with the mean equals to the final result.

Let's first consider the *sum* and *count* queries. For $x_i \in T_1$, we define $\mu(x_i, T_1)$ as

$$\mu(x_i, T_1) = \sum_{\forall y_j \in T_2} |T_1| \text{expression}(x_i, y_j) \quad (8.3)$$

where $|X|$ and $|Y|$ denote the number of retrieved samples for T_1 and T_2 respectively, and $\text{expression}(x_i, y_j)$ is the aggregation operator for the join result of x_i and y_j (if x_i cannot join with y_j or the join result does not satisfy some the predicates, $\text{expression}(x_i, y_j) = 0$). As we can see, the average value of $\mu(x_i, T_1)$ for all $x_i \in T_1$ is equal to the final result μ . Let σ_x^2 denote the variance computed for T_1 in the join operation. We have

$$\sigma_x^2 = \frac{\sum_{\forall x_i \in T_1} (\mu(x_i, T_1) - \mu)^2}{|T_1|} \quad (8.4)$$

Similarly, we can compute $\mu(y_j, T_2)$ and σ_y^2 for table T_2 . Then, the variance of the join operation is computed as

$$\sigma^2 = \frac{\sigma_x^2}{|T_1|} + \frac{\sigma_y^2}{|T_2|} \quad (8.5)$$

In online aggregation, suppose X and Y are seen samples from T_1 and T_2 , respectively. We use X and Y to replace T_1 and T_2 respectively in above definitions. Namely, we have

$$\bar{\sigma}_x^2 = \frac{\sum_{\forall x_i \in X} (\mu(x_i, X) - \bar{\mu})^2}{|X|} \quad (8.6)$$

and

$$\bar{\sigma}^2 = \frac{\bar{\sigma}_x^2}{|X|} + \frac{\bar{\sigma}_y^2}{|Y|} \quad (8.7)$$

where $\bar{\mu}$ is the result estimated from the sample sets. Applying the central limit theorem, we can compute the error bound and confidence correspondingly.

Compared to the *sum* and *count* queries, estimating the variance of *avg* queries are more complex. *avg* can be considered as a combined operator (*sum* divided by

count). Given $x_i \in T_1$ and $y_j \in T_2$, let $f(x_i, y_j)$ return 1, if x_i can join with y_j and the result satisfies the predicates. Otherwise $f(x_i, y_j)$ returns 0. Given a record x_i in T_1 , we define another function $\mu'(x_i, T_1)$ as

$$\mu'(x_i, T_1) = \sum_{\forall y_j \in T_2} |T_1| f(x_i, y_j) \quad (8.8)$$

To estimate the average value, we need to compute the covariance of $\mu(x_i, T_1)$ and $\mu'(x_i, T_1)$. Specifically, let x_{avg} and x'_{avg} denote the average values of $\mu(x_i, T_1)$ and $\mu'(x_i, T_1)$, respectively. The covariance is computed as:

$$\gamma(x_i) = \frac{1}{|T_1|} \sum_{\forall x_i \in T_1} (\mu(x_i, T_1) - x_{avg})(\mu'(x_i, T_1) - x'_{avg}) \quad (8.9)$$

The above definitions are applied to T_2 as well. The covariance of join results is estimated as

$$\gamma = \frac{\gamma(x_i)}{|T_1|} + \frac{\gamma(y_j)}{|T_2|} \quad (8.10)$$

And finally, we compute the variance of the join results as

$$\sigma^2 = \frac{1}{\mu_c^2} (\sigma_s^2 - 2\mu\gamma + \mu^2\sigma_c^2) \quad (8.11)$$

where μ_c and μ_s denote the *count* and *sum* results respectively, $\mu = \frac{\mu_s}{\mu_c}$, σ_c^2 and σ_s^2 are variances computed for *count* and *sum* queries respectively. As before, in the real processing, the sample sets X and Y are used to replace T_1 and T_2 in the computations.

One problem of ripple join is its extensive usage of memory. When the memory is insufficient to hold the tables, it degrades to block ripple join, which is quite inefficient. In [15], a parallel hash-based ripple join is proposed to address the problem. The basic idea is to partition the tuples among the processors and each processor can process the ripple join individually. The final result is obtained by combining all the estimations. In [12], a disk-based ripple join is used to handle the memory overflow problem. The disk-based ripple join has several phases. Each phase contains a set of hash-based ripple join in memory. It maintains the statistical confidence from the start-up through the completion. Interested readers can refer to the papers for more details.

8.3.2 Online Aggregation for Multi-query Processing

In [21], online aggregation is used for multi-query processing. Multi-query optimization is well studied in real systems, as

1. The system is designed to support multiple users, who can issue queries concurrently.

2. A single complex query can be decomposed into multiple sub-queries. For example, a nested query involves aggregates in both the outer and inner query blocks. Such queries always operate on the same set of tables, which can be optimized together.
3. A user can roll up and drill down in the data cube, which results in multiple queries being submitted to the system.

Optimizing multiple queries can potentially improve the performance significantly. For example, consider the following three queries (using TPC-H schema as the example):

- Q_1 select avg(discount) from lineitem where quantity < 20 group by returnflag;
 Q_2 select avg(discount) from lineitem where returnflag='r' or returnflag='a';
 Q_3 select avg(discount) from lineitem;

Suppose Q_1 is submitted to the system and when it is being processed, Q_2 is submitted, followed by Q_3 . All three queries are processed against table *lineitem*. Every tuple in *lineitem* is a valid sample for Q_3 , while a portion of tuples can be used to process Q_1 and Q_2 . Instead of processing the queries individually, we can process them together. Given a tuple t in *lineitem*, it is firstly used to process Q_3 . If it satisfies the condition *quantity* < 20, we use it to update the result of Q_1 . If it also has the *returnflag* as 'r' or 'a', we use it to compute Q_2 as well. In this way, a sample is shared among multiple queries and we reduce the I/O costs.

However, sharing samples among queries are not easy. A valid sample for one query is not necessarily applicable to another query. As queries join and leave the system, the sharing strategy changes dynamically. We need to compute the overlap between different queries, which are costly. Moreover, in some queries, a sub-query is evaluated repeatedly, which can be avoided. To address the above issues, in [21], two techniques are applied, scrambling and space partitioning.

Figure 8.6 shows the general idea of online aggregation for multiple queries. In the preprocessing phase, a data scrambler is employed to permute the original

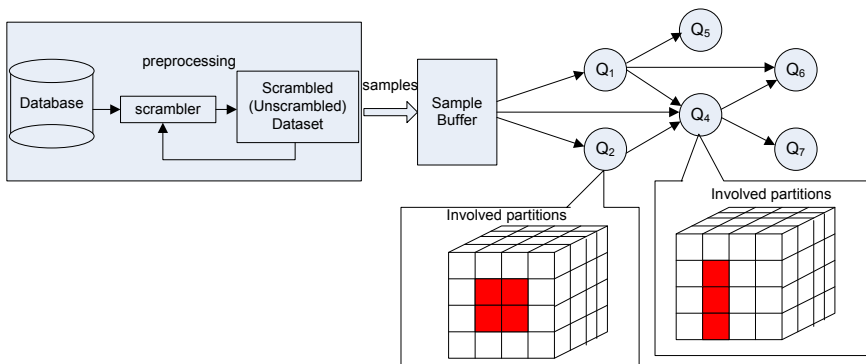


Fig. 8.6 Architecture of Online Aggregation for Multi-Queries.

dataset iteratively. The result is stored as a scrambled dataset. By scanning the scrambled dataset, a stream of random samples is generated and used to process queries. At runtime, the queries are organized into a dissemination graph based on some partitioning strategy. Both the samples and partial aggregation results are shared between linked queries. In the following discussion, we briefly introduce the implementation of the data scrambler and space partitioning strategy.

8.3.2.1 Data Scrambler

Scrambling is used to generate a stream of samples in a low-cost way. Traditional scheme picks samples randomly from the dataset at runtime, which may incur high overhead. To solve this problem, the dataset is randomly scrambled in the preprocessing phase. By scanning the scrambled dataset, a stream of random samples is generated. This strategy has two advantages. First, in conventional scheme, we get one sample from a random page, while in scrambled dataset, one page of records can be used as samples. Second, in scrambled dataset, sequential scan is used to generate sample stream, which is much more efficient than random access in other approaches. Completely random disk access can be five orders of magnitude slower than sequential access [11].

To scramble a dataset, a simple strategy can be applied. In particular, the data file is scanned sequentially; as we scan each tuple, we place it in a randomly picked position in a scrambled dataset (which is initially empty). The scrambled dataset is then used as the input and we repeat the above process several times to fully permute the records. Another approach is relying on the *RAND* function of DBMS². Suppose we try to scramble table *T*. We alter the table by adding a new column, *rowid*.

- ALTER TABLE *T* ADD *rowid* int;

Then, we fill the *rowid* with random integers.

- UPDATE *T* SET *rowid* = *RAND*() × *C* WHERE always true condition;

C is a constant, defining the range of *rowid* ($[0, C)$). In the WHERE clause, we use an always true condition to update the values of all *rowids*. Then, a new table is created by sorting the old table.

- INSERT INTO TABLE *T'* (Column 1, Column 2,..., Column *k*) SELECT * FROM *T* ORDER BY *rowid* ASC;

The new table *T'* is created by copying the old table *T*. But we permute the sequence of records by sorting via the random values of *rowid*. To further scramble the table, the above process can be repeated multiple times.

To process join operation, we may need to build scrambled datasets for multiple tables. Depending on the applications, two approaches can be applied. In a data warehouse system, we can precompute the join result of dimensional tables and the fact table. And the result table is scrambled to process queries. Alternatively,

² DBMSs implement random functions differently. In our discussion, we use MySQL as the example.

if storage cost is crucial, we just scramble each table and use the index to retrieve random samples. In particular, we build indexes on the join attributes of queries. To generate a random sample stream from a two-way join, we pick one relation as the outer relation (R) and the other relation as the inner relation (S). R is scanned to retrieve random samples and for each sample record from R , we search the index of S to generate random join results.

When local datasets are updated, we need to rebuild the scrambled datasets. Fortunately, in data warehouse system, updates are processed in a batch manner. To update the scrambled dataset, we can

- Totally rebuild the scrambled datasets, if a large number of updates need to be performed.
- For each newly inserted tuple t , we randomly select a tuple t' from scrambled dataset and replace t' with t . t' is appended to the end of the scrambled dataset.

When the first strategy is applied, the query processing is stopped and will resume after new scrambled datasets are created.

8.3.2.2 Space Partitioning

To facilitate the sample and result sharing among queries, the space is partitioned adaptively. Figure 8.7 illustrates the idea. Suppose two queries Q_1 and Q_2 are submitted to the system and they overlap with each other. We can partition the data space into 5×5 partitions by the boundaries of the queries. This strategy allows Q_1 and Q_2 to share a common grid, where samples or even the aggregation results can be reused.

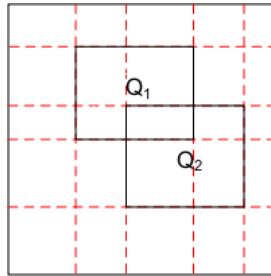


Fig. 8.7 Query-based Partitioning.

The space partitioning scheme works as follows. First, we collect a set of queries S_q . Those queries represent the query patterns (we assume the query pattern changes infrequently). For $q \in S_q$, we transfer q into a set of range predicates, one for each attribute. Then, the partitioning scheme in Figure 8.7 is applied to partition the space. Namely, we split the range of an attribute based on the range predicates of queries. In this way, a set of grids are generated. For each grid g_i , it is fully contained by

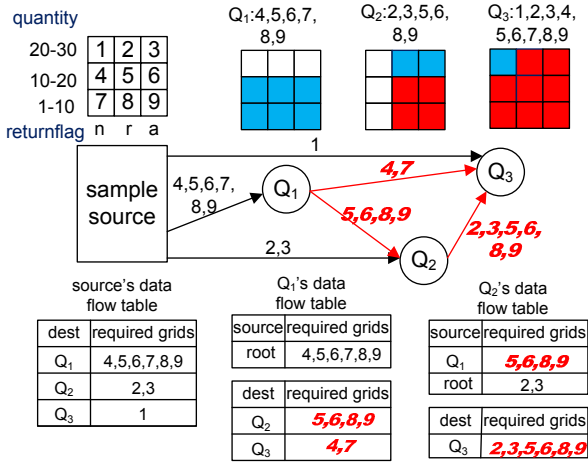


Fig. 8.8 Salvaging Results of Queries.

some queries. Thus, we can group the samples by the grids they belong to and compute aggregate results for each grid. However, the above approach may generate too many grids, which may lead to high maintenance overhead. To handle this problem, a ranking strategy is provided. We rank the grids by the number of overlapped queries. And adjacent grids with low ranks are combined to reduce the total number of grids.

After partitioning, the queries are organized as a dissemination graph to share samples. Each query acts as a node in the dissemination graph and the sample source is the root node. Specifically, the root node scans the scrambled data repeatedly to generate a stream of random samples. If there are other query nodes in the graph, the root will forward the samples to the corresponding query nodes. Figure 8.8 illustrates the idea of dissemination graph and how to reuse samples in the graph. Suppose the table is partitioned by *quantity* and *returnflag* into 9 grids. We have three queries, Q_1 , Q_2 and Q_3 . Suppose Q_1 first comes to the system, followed by Q_2 and Q_3 . Q_1 overlaps with grids 4, 5, 6, 7, 8 and 9; Q_2 involves grids 2, 3, 5, 6, 8 and 9; and Q_3 covers the whole data space. We construct the dissemination graph by examining the common grids between queries.

When a query is inserted, we search the dissemination graph to find existing queries that can be exploited to share samples. A greedy strategy is applied to maximize the number of shared grids. It works progressively by selecting the query that overlaps with the incoming query mostly in each iteration. Then, the two queries create a link and the shared grids are removed from consideration in the next iteration. This progress continues, until no more grids can be shared. The query will create a link to the root node if some of its grids cannot be shared.

In Figure 8.8, when Q_1 joins the system, no other queries exist. Therefore, it directly retrieves samples from the sample source. So it creates a link with the root node. When Q_1 is being processed, Q_2 is submitted. It finds that it can share grid 5,

6, 8 and 9 with Q_1 . Hence, Q_2 is linked with Q_1 . As not all grids can be obtained from Q_1 , Q_2 also connects to the root node for grids 2 and 3. Finally, when Q_3 comes to the system, except grid 1, all other grids of Q_3 can be inherited from Q_1 and Q_2 .

In the dissemination graph, each node keeps records of how samples are shared. For example, in Figure 8.8, Q_1 keeps two tables. One table lists the incoming sample flows. Since Q_1 receives samples from the root node, there is only one record in the table. The other table keeps the track of outgoing sample flows. In Q_1 , two records are created for Q_2 and Q_3 , respectively.

When a query completes (e.g., terminated by user or the whole scrambled dataset has been scanned), it leaves the dissemination graph, resulting a restructuring process. We link its successors with its predecessors. For example, suppose Q_1 leaves the dissemination graph in Figure 8.8, Q_1 's successors, Q_2 and Q_3 , will retrieve grids 4, 5, 6, 7, 8 and 9 from the root instead of Q_1 .

8.3.2.3 Query Processing

A straight-forward way to process multiple queries is to stream samples to the queries based on the dissemination graph. Each query, when receiving a set of samples, will recompute its results and confidence. This strategy shares the samples among the queries. We can apply a more aggressive sharing strategy. In particular, we compute the aggregation result for each grid and share the results among the queries.

Given a query Q_i and a grid g_j , based on the relationship between Q_i and g_j , different processing strategies are used. If g_j partially overlaps with Q_i or no existing queries fully cover g_j , Q_i will directly get samples of g_j from the root, as in this case, no aggregation results can be shared. Otherwise, Q_i must create a link to an existing query, which fully covers g_j . Instead of computing the aggregation result for g_j , Q_i will reuse the results computed by other queries. Thus, in the dissemination graph, there are two types of links. Links between root node and query nodes transfer the samples, while links between queries transfer the aggregation results.

To support the above processing strategy, we need to modify the original online aggregation algorithm.

1. We compute aggregation results for the grids involved in the query processing (the grids that register at the outflowing table of the root node). For grid g_j , after receiving a sample, it will update its estimated aggregation results (e.g., *avg*, *count*, *sum* and etc.). For each estimation, the grid also keeps the variance. Variances are used to generate the confidence and error bound. Note that g_j only needs to be computed once, even it may appear in multiple queries. In Figure 8.8, grid 5 is computed at Q_1 and then the results are shuffled to Q_2 and Q_3 .
2. The grid updates its estimation individually. The query engine can terminate the computation of a specific grid. Then, no sample is retrieved for the grid. Each grid also keeps track of which samples have been used (by recording the block pointer of the first seen sample). If all samples in the scrambled dataset are used up, the grid stops its processing with precise results.

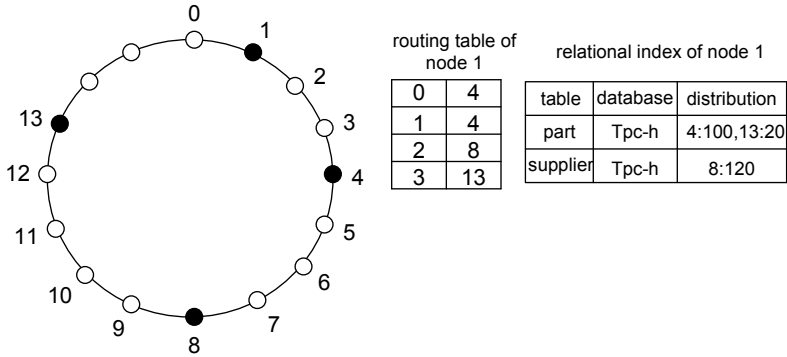


Fig. 8.9 Chord Overlay.

3. To answer a query, results from multiple grids need to be merged. We assign a weight to each grid's estimation based on the data distribution. The variances of grids are also combined to generate the confidence for the merged result.

The statistical model is similar to the conventional online aggregation. Interested readers can refer to the paper [21].

8.3.3 Distributed Online Aggregation

Online aggregation can be also applied in the distributed database systems, Peer-to-Peer systems or other distributed systems. A simple solution is to forward the query to the corresponding nodes, where a conventional online aggregation algorithm is applied. The approximate results are then sent back and combined. By parallelizing the computation, the solution can provide a better performance. But the challenge is handling queries involving multiple tables in different nodes. In that case, data need to be shuffled between nodes, which incurs significantly high overhead. To address this problem, [20] adopts an adaptive processing strategy. In this section, we give a brief overview of the idea.

The core techniques in [20] are DHT (Distributed Hash Table) and linear hash function [14]. DHT is used to partition the data, while LSH is used to dynamically tune the number of nodes involved. Before delving into the online aggregation issues, we first review the two related techniques.

In [20], Chord [18] is used as the DHT to illustrate the idea. Figure 8.9 shows a Chord ring with 16 nodes. The dark nodes denote the real compute nodes, while the white nodes are the possible keys. In Figure 8.9, we have four compute nodes with keys 1, 4, 8 and 13. Each compute node in the overlay maintains successor and predecessor links. E.g., node 4 and node 13 are the successor and predecessor nodes of node 1, respectively. Each node is responsible for a key range, starting from its predecessor's key to its own key. E.g., the key range of node 4 is (1, 4]. Given a query for a specific key, we can follow the successor/predecessor links to route the query. To speed up the search, each node also maintains $\log_2 N$ routing neighbors,

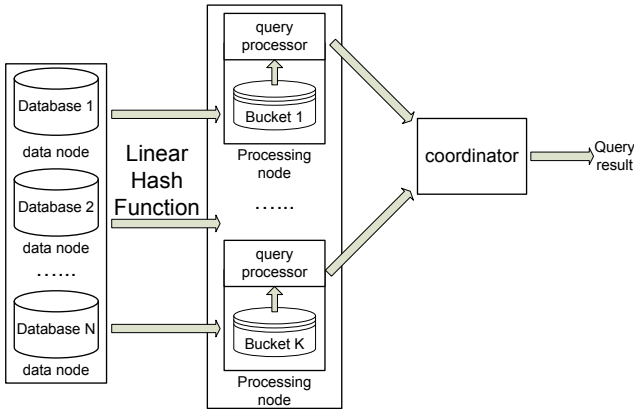


Fig. 8.10 Data Flow.

where N is the number of nodes in Chord. Specifically, the routing neighbor has 2^x ($0 \leq x \leq \log_2 N$) distance to the owner node. By applying the routing neighbors, the query can be completed within $O(\log_2 N)$ hops.

In normal hash function, a predefined number of buckets are used and the collision is solved by linked lists. However, searching the linked lists is costly. Therefore, linear hashing is proposed to dynamically increase the number of buckets. Under linear hashing, a hash function adaptively doubles the number of buckets in each cycle; however, the physical buckets are only allocated one at a time whenever an existing bucket overflows. Whenever a physical bucket is created, the data in the corresponding bucket are rehashed, and redistributed between the two buckets. We will show a simple example of linear hashing to demonstrate how the distributed online aggregation works.

Figure 8.10 shows the data flow of the distributed online aggregation. Each node in the system maintains its own database individually. And the nodes join a Chord ring to collaborate for query processing. To process a query, samples are retrieved from the databases and disseminated by a linear hash function. A unique namespace is generated for each bucket of the linear hash function. And we use the namespace of a bucket as the key to publish it in the Chord ring. In this way, all samples of the same bucket will be sent to the same node for processing. Linear hashing is adopted to dynamically adjust the number of involved nodes. When the query needs to scan a large dataset, more nodes will join the processing. In each node, we apply the conventional online aggregation algorithm to generate an approximate result. And all the approximate results are sent to a coordinator, where the final result is computed by merging all the partial results.

8.3.3.1 Distributed Sampling

To generate local samples, the node can invoke sampling approaches discussed in the previous section. However, combining random samples from different node does

not lead to a random sampling result, as data may follow skewed distribution. For example, suppose nodes n_1 , n_2 and n_3 have 10k, 20k and 30k tuples of table T . If we combine the samples from n_1 , n_2 and n_3 uniformly, n_1 's data actually have a higher probability of being sampled than those of n_2 and n_3 . Therefore, an alternative way is to retrieve samples from nodes adaptively. In particular, given a table T , suppose P nodes contain tuples of T . Let the set of tuples of T at node n_i be T_i ($1 \leq i \leq P$). Each node n_i provides the number of samples that is proportional to its table size, i.e., node n_i provides $(\frac{|T_i|}{|T|} \cdot k)$ samples, where $|T|$ refers to the size of table T .

To perform the above sampling, we need the statistics about the number of records in each table. We can build distributed histograms to collect the statistics. But to save cost, the histograms cannot be updated frequently. Therefore, we may get stale data distribution information. To correct this, the sampling is performed in two phases. In the first phase, the samples are retrieved from each node based on the histograms. The node returns both the samples and the size of the corresponding table. In the second phase, the sampling process adjusts its strategy based on the returned table sizes from nodes. By combining the samples of two phases, we can get a random sample set.

8.3.3.2 Sample Dissemination

When receiving the sampling request, the node publishes its samples based on the Chord protocol. Samples of the same bucket are sent to the same node for processing by using the bucket ID as the key. Linear hashing is used to dynamically adjust the number of involved nodes (e.g., if more samples are required, linear hashing will increase the number of buckets). When more nodes participant in the query processing, the query will get a short response time due to the parallelism.

Figure 8.11(a) to Figure 8.11(d) illustrate how samples are disseminated in the network. Suppose, we have three databases (DB1, DB2 and DB3) maintained by three data nodes. To process a query, 8 samples are required. Hence, we will retrieve 4, 2 and 2 samples from DB1, DB2 and DB3, respectively. Suppose the size of the bucket in linear hashing is 2 and the hash function is defined as $h(k)=k \bmod 2^i$, where i is the level of the linear hash. In Figure 8.11(a), two samples 8 and 9 are published. At first, there is only one bucket and all data are inserted into the bucket. After 8 and 9 are inserted, the bucket is full and cannot accept more samples. When new value 3 is inserted, the bucket splits and increases its level by 1. Now, the hash function becomes $h(k)=k \bmod 2$. And thus, 8 is kept in B_0 and 9 and 3 are stored in B_1 .

When new value 5 is inserted into B_1 , B_1 becomes overloaded as shown in Figure 8.11(b). However, we cannot split it as the level pointer is set to B_0 . In Figure 8.11(c), after samples 6 and 10 are inserted, B_0 becomes overloaded and splits half of its data to the new bucket B_2 based on the hash function $h(k)=k \bmod 4$. The level of B_0 increases by 1 and the level pointer moves to B_1 . As B_1 already satisfies the split condition, it creates the new bucket B_4 and increases its level by 1. And the level pointer is reset to B_0 . In the end, Figure 8.11(d) shows the final status of the buckets.

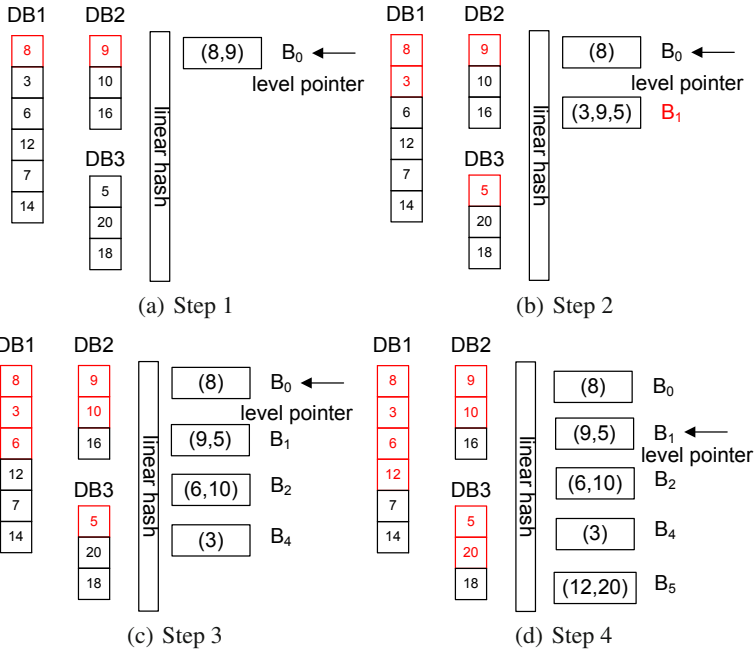


Fig. 8.11 Dissemination of samples.

8.3.3.3 Result Combination

In each bucket, the conventional online aggregation scheme is applied to process the query. And an approximate result is generated. To combine the results of different buckets, a node is selected as the coordinator. The buckets send their partial results to the coordinator periodically, where a final approximate result is created. Suppose k buckets are used and X_i is the result of bucket b_i 's estimation. Let σ_{X_i} denote the variance of X_i and $Cov(X_i, X_j)$ represent the covariance of X_i and X_j . The variance for the final (approximate) result is estimated as:

$$\sigma = \sum_{i=1}^k w_i \sigma_{X_i} + \sum_{i=1}^k \sum_{j=1, j \neq i}^k Cov(X_i, X_j) \tag{8.12}$$

We assign a weight w_i to each individual estimation ($\sum_{i=1}^k w_i = 1$). Based on the analysis of [12], the covariance can be ignored in most cases. And hence,

$$w_i = \frac{1}{\sigma_{X_i} \sum_{j=1}^k \frac{1}{\sigma_{X_j}}} \tag{8.13}$$

And thus, the final result is computed as:

$$X = \sum_{i=1}^k w_i X_i \quad (8.14)$$

Because the sum of normal distributed variants follows normal distribution as well, we can use the same rule to estimate the confidence and error bound of the result. The coordinator updates the approximate results for the user. And if the user is satisfied with the results, the coordinator can terminate the query processing by stopping the sampling process.

8.3.3.4 Processing Multi-relational Query

To process queries involving join operations, samples need to be shuffled between nodes. However, a good sample dissemination scheme can reduce the overhead. For example, given query $T_1 \bowtie_{T_1.a=T_2.b} T_2$, we can use the values of $T_1.a$ and $T_2.b$ as the keys in the linear hash function. In this way, $\forall t_i \in T_1 \forall t_j \in T_2$, if t_i can join with t_j , t_i and t_j must be mapped to the same bucket of the linear hash function. Therefore, we can apply the ripple join algorithm in each bucket individually to compute the approximate results.

In a more complex case, suppose we need to process query $T_1 \bowtie_{T_1.a=T_2.b} T_2 \bowtie_{T_2.c=T_3.d} T_3$, we have two approaches. First, we can disseminate samples of T_1 and T_2 based on $T_1.a$ and $T_2.b$. Samples of T_3 are published by using $T_3.d$ as the key. And then, the buckets for T_1 and T_2 use $T_2.c$ to re-distribute the samples to the buckets of T_3 , where the ripple join algorithm can be applied. Or alternatively, we can first group samples of T_2 and T_3 based on $T_2.c$ and $T_3.d$. And then the buckets rehash the samples to the buckets of T_1 . In either way, we need to guarantee that joinable samples are hashed to the same bucket.

8.3.4 Online Aggregation and MapReduce

MapReduce [8] is proposed by Google as a programming framework to process large-scale data analytical jobs in clusters. It soon becomes popular, due to its flexibility, scalability and fault tolerance. In MapReduce, two interfaces, *map* and *reduce*, are defined. *map* reads data from distributed file system (DFS), database system or other storage systems and transforms the data into a set of key-value pairs. Those key-value pairs are grouped in the *reduce* function, where user-defined processing logic is applied to process values of the same key. Finally, the results are written back to the underlying storage system, e.g., DFS. Figure 8.12 shows the data flow of MapReduce. As both *map* and *reduce* processes can run on multiple nodes independently, MapReduce achieves its efficiency by the parallelism.

MapReduce is originally designed for batch processing. The *reduce* phase is strictly after the *map* phase. This design is different from most query processing strategies in the database systems, where pipeline is widely used. In [6], a

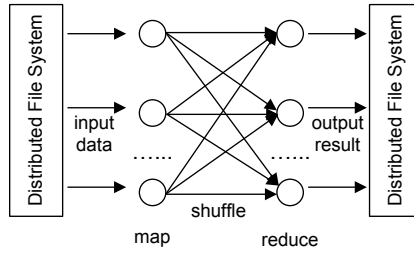


Fig. 8.12 Data Flow in MapReduce.

modified MapReduce framework is proposed, which allows data to be pipelined between the operators. On top of the new framework, online aggregation technique can be supported seamlessly [7].

The pipelining approaches in [6] can be classified as intra-job pipeline and inter-job pipeline.

Intra-job Pipeline. A MapReduce job is processed in two phases, the *map* phase and the *reduce* phase. Between the phases, data need to be shuffled from the *map* process to the *reduce* process. In conventional MapReduce, *map* process will write its output (a set of key-value pairs) into local disk. The *reduce* process pulls the output files of the *map* processes. Only when all *map* processes complete their processing, will the *reduce* processes start the work. To address this problem, instead of writing the output to local disk, the *map* process directly pushes its results to the *reduce* process. And the *reduce* process can start its processing as soon as possible.

Inter-job Pipeline. A complex analytical query is always translated into a set of MapReduce jobs, which are processed one by one. When a job completes, it writes back all the results back to DFS. The next job will read the results of the last job to continue the processing. This strategy incurs high I/O costs. Therefore, in [6], the *reduce* process of a job can directly pipeline its result to the *map* processes of the next job. This strategy avoids the repeated read/write operations. However, fault tolerance is compromised, because partial results may not be available in DFS. A solution is to add checkpoints after some jobs. When processing reaches a checkpoint, the system backs up the results into DFS.

To implement the online aggregation in MapReduce framework, users need to customize their *reduce* functions. As *reduce* processes run independently of each other, it is difficult to provide a correct estimation about the accuracy. Instead, in [6], progress (percent of processed data) is used as the indicator on how good the result is. In particular, the *reduce* process will generate a result for predefined progresses, e.g., 10%, 20% and 30% of input data. The results are written back to DFS and can be identified by the file names. User's application can search the DFS to get results of a specific progress, e.g., 10%. By combining the results from all *reduce* process, the application can provide an approximate estimation. In [6], online aggregation for multiple MapReduce jobs is also discussed. Interested readers can refer to the paper for more details.

8.4 Conclusion and Discussion

As a promising technique, online aggregation has attracted much interests from many researchers. It can significantly reduce the processing cost of aggregate queries. In a decision making system that requires fast response, online aggregation can be applied to provide real-time approximate results. However, up to now, no commercial database supports this function. This is probably because

- It is challenging to provide a good estimation for skewed data distribution. When data are skewed, conventional sampling approach cannot work well. The samples may not be randomly distributed. And therefore, the generated results are more affected by the dense parts.
- Some “group by” queries may lead to unexpectedly long processing time because of a small number of samples. For example, given 100k records of students, suppose we try to compute the average score of each class. If a class A is selected by 10 students, the next sample comes from class A with probability 0.0001. Thus, the query processor needs to retrieve a large number of samples, until it can provide a good enough estimation for class A.
- Handling updates in online aggregation is not a trivial problem. As online aggregation also guarantees to provide a precise result after scanning the whole dataset, the updates actually affect the sampling process. When online aggregation stops with a precise result, all existing tuples need to be sampled exactly once. Moreover, updates may invalid the previous estimation as data distribution changes.

Addressing the above problems can facilitate the adoption of online aggregation. In fact, in real systems, queries are generated by the templates (e.g., queries submitted by the web form). Online aggregation can be specially designed for a subset of queries, which is easier to optimize. Moreover, online aggregation can also be used to answer the continuous queries in the streaming system, where query response time is very important and samples are received at varied rates.

References

1. TPC-H Benchmark, <http://www.tpc.org/tpc-h>
2. Acharya, S., Gibbons, P.B., Poosala, V.: Congressional Samples for Approximate Answering of Group-By Queries. In: SIGMOD Conference, pp. 487–498 (2000)
3. Acharya, S., Gibbons, P.B., Poosala, V., Ramaswamy, S.: Join Synopses for Approximate Query Answering. In: SIGMOD Conference, pp. 275–286 (1999)
4. Babcock, B., Chaudhuri, S., Das, G.: Dynamic Sample Selection for Approximate Query Processing. In: SIGMOD Conference, pp. 539–550 (2003)
5. Chaudhuri, S., Das, G., Datar, M., Motwani, R., Narasayya, V.R.: Overcoming Limitations of Sampling for Aggregation Queries. In: ICDE, pp. 534–542 (2001)

6. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce Online. Tech. rep., University of California, Berkeley (2009), <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-136.pdf>
7. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Gerth, J., Talbot, J., Elmeleegy, K., Sears, R.: Online Aggregation and Continuous Query Support in MapReduce. In: SIGMOD Conference, pp. 1115–1118 (2010)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI, pp. 137–150 (2004)
9. Haas, P.J., Hellerstein, J.M.: Ripple Joins for Online Aggregation. In: SIGMOD Conference, pp. 287–298 (1999)
10. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online Aggregation. In: SIGMOD Conference, pp. 171–182 (1997)
11. Jacobs, A.: The Pathologies of Big Data. *Commun. ACM* 52(8), 36–44 (2009)
12. Jermaine, C., Dobra, A., Arumugam, S., Joshi, S., Pol, A.: A Disk-Based Join With Probabilistic Guarantees. In: SIGMOD Conference, pp. 563–574 (2005)
13. Jermaine, C., Pol, A., Arumugam, S.: Online Maintenance of Very Large Random Samples. In: SIGMOD Conference, pp. 299–310 (2004)
14. Litwin, W.: Linear Hashing: A New Tool for File and Table Addressing. In: VLDB, pp. 212–223 (1980)
15. Luo, G., Ellmann, C.J., Haas, P.J., Naughton, J.F.: A Scalable Hash Ripple Join Algorithm. In: SIGMOD Conference, pp. 252–262 (2002)
16. Olken, F.: Random Sampling from Databases. Ph.D. thesis. University of California (1993)
17. Olken, F., Rotem, D.: Maintenance of Materialized Views of Sampling Queries. In: ICDE, pp. 632–641 (1992)
18. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: SIGCOMM, pp. 149–160 (2001)
19. Tan, K.L., Goh, C.H., Ooi, B.C.: Online Feedback for Nested Aggregate Queries with Multi-Threading. In: VLDB, pp. 18–29 (1999)
20. Wu, S., Jiang, S., Ooi, B.C., Tan, K.L.: Distributed Online Aggregation. *PVLDB* 2(1), 443–454 (2009)
21. Wu, S., Ooi, B.C., Tan, K.L.: Continuous Sampling for Online Aggregation over Multiple Queries. In: SIGMOD Conference, pp. 651–662 (2010)

Chapter 9

Adaptive Query Processing in Distributed Settings

Anastasios Gounaris, Efthymia Tsamoura, and Yannis Manolopoulos

Abstract. In this survey chapter, we discuss adaptive query processing (AdQP) techniques for distributed environments. We also investigate the issues involved in extending AdQP techniques originally proposed for single-node processing so that they become applicable to multi-node environments as well. In order to make it easier for the reader to understand the similarities among the various proposals, we adopt a common framework, which decomposes the adaptivity loop into the monitoring, analysis, planning and actuation (or execution) phase. The main distributed AdQP techniques developed so far tend to differ significantly from their centralized counterparts, both in their objectives and in their focus. The objectives in distributed AdQP are more tailored to distributed settings, whereas more attention is paid to issues relating to the adaptivity cost, which is significant, especially when operators and data are moved over the network.

9.1 Introduction

The capability of database management systems to efficiently process queries, which are expressed as declarative statements, has played a major role in their success over the last decades. Efficiency is guaranteed due to several sophisticated optimization techniques, which heavily rely on the existence of metadata information about the data to be processed, such as the distribution of values and the selectivity of the relational operators. Since the late 1970s and the introduction of System R [58], static optimization of query plans and subsequent execution has been the main choice for database system developers. However, when the metadata required are not available or accurate at compile time, or when they change during execution,

Anastasios Gounaris · Efthymia Tsamoura · Yannis Manolopoulos
Aristotle University of Thessaloniki, Thessaloniki, Greece
e-mail: {gounaria, etsamour, manolopo}@csd.auth.gr

the query processor needs to revise the current execution plan on the fly. In this case, query processing is called adaptive.

In adaptive query processing (AdQP), there is a feedback loop, similar to the one appearing in autonomic systems, according to which the query processor monitors its execution properties and its execution environment, analyzes this feedback, and possibly reacts to any changes identified with a view to ensuring that either the current execution plan is the most beneficial or a modification of the current plan can be found that is expected to result in better performance.

Although AdQP is particularly relevant to wide area settings, in which query statistics are more likely to be limited or potentially inaccurate, and the computational properties, such as the processing capacity of hosting machines, are volatile, most AdQP proposals have focused either on completely centralized query processing or on centralized processing of data retrieved or stemming from remote sources and data streams, respectively. In such settings, there is typically a single physical machine used for query execution, which is predefined, and thus the focus is mostly on adapting to changing properties of the data processed, e.g., cardinalities of intermediate results and operator selectivities. This is, of course, of high importance for distributed query processing (DQP), as crucial information about the data may be missing at compile time. However, of equal significance are adaptations to changing properties of a potentially arbitrary set of resources that DQP may employ and of their communication links. Currently, AdQP with respect to changing resources is not addressed as satisfactorily as with respect to changing data properties.

In this survey chapter, we systematically discuss AdQP techniques that are tailored to distributed settings both with respect to the data sources and the processing nodes. We also investigate the issues involved in extending AdQP techniques originally proposed for single-node processing so that they become applicable to multi-node environments as well. In order to make it easier for the reader to understand the similarities among the various proposals, we adopt a common framework, which decomposes the adaptivity loop into its constituent phases mentioned above, i.e., monitoring, analysis, planning and actuation phase. The later corresponds to the phase, in which the adaptivity decisions are executed by the system.

Structure. The structure of this chapter is as follows. In the remainder of this section we briefly discuss preliminary concepts of distributed query processing and optimization (Section 9.1.1), and related work (Section 9.1.2). In Section 9.2 we present the framework that forms the basis of our analysis. The next section contains a short review of traditional AdQP for centralized settings and explains the reasons why such techniques cannot be applied to wide-area environments in a straightforward manner. The discussion of the AdQP techniques for distributed settings, which is the core part of this chapter, is in Sections 9.4-9.6. Existing work in distributed AdQP techniques can be classified in three broad categories. Techniques that do not rely on the existence of traditional query plans fall into the first category, which is examined in Section 9.4. The second category comprises approaches that perform load management at the operator level (Section 9.5), whereas, in Section 9.6 we discuss distributed AdQP techniques where the adaptivity occurs at a higher level.

The final section (Section 9.7) contains an assessment of the current status in the area, along with directions for future work, and concludes the chapter.

9.1.1 Distributed Query Processing Basics

Distributed query processing consists of the same three main phases as its centralized counterpart, namely parsing (or translation), optimization and execution. During parsing, the initial query statement, which is expressed in a declarative language such as SQL, is translated into an internal representation, which is the same for both centralized and distributed queries [42].

Query optimization is commonly divided into two parts, query rewriting and cost-based plan selection. Query rewriting is typically carried out without any statistical information about the data and independently of any previous physical design choices (e.g., data locations, existence of indices) apart from the information about data fragments. In distributed queries over non-replicated fragmented data, the relevant data fragments are identified during this procedure as well [51]. Secondly, cost-based optimization is performed. The search strategy typically follows a dynamic programming approach for both centralized and distributed queries [43, 46] provided that the query is not very complex in terms of the number of different choices that need to be examined; in the latter case the plan space is reduced with the help of heuristics. Traditional cost based optimization is capable of leading to excellent performance when there are few correlations between the attributes, adequate statistics exist and the environment is stable.

The optimized plan is subsequently passed on to the query execution engine, which is responsible for controlling the data flow through the operators and implementing the operators. Although in both traditional disk-based queries and continuous queries over data streams the operators are typically those that are defined by the relational algebra (or their modifications [69]), the execution engine may differ significantly. In disk-based queries, the pull-based iterator model of execution is preferable [31], according to which each operator adheres to a common interface that allows pipelining of data, while explicitly defining the execution order of query operators and ensuring avoidance of flooding the system with intermediate results in case of a bottleneck. On the other hand, continuous queries over data streams may need to operate in a push-based mode [8]. The main difference between the push and pull model of execution lies in the fact that, in the push model, the processing is triggered by the arrival of new data items. This property may give rise to issues that are not encountered in pull-based systems, which have full control on the production rate of intermediate results. For example, push-based query processors may need to resort to approximation techniques when data arrival rates exceed the maximum rate in which the system can process data. We do not deal with approximation issues in this chapter; we refer the interested reader to Chapters 7 and 10 of this book. However, it is worth mentioning that AdQP in push-based systems considers additional issues, such as adaptations to the data arrival rates.

In conventional static query processing, the three phases of query processing occur sequentially, whereas, in AdQP, query execution is interleaved with query optimization in the context of a single query with a view to coping with the unpredictability of the environment, and evolving or inaccurate statistics. According to a looser definition of AdQP, the feedback collected from the query execution of previous queries impacts on the optimization of future queries (e.g., [61]); we do not deal with such flavors here. Note that the need for on-the-fly re-optimizations is mitigated with the help of advanced optimization methodologies, such as robust optimization (e.g., [3, 12]). Also, other topics related to AdQP are discussed in Chapter 10 (on combining search queries and AdQP).

9.1.2 *Related Work*

A number of surveys on AdQP have been made available [35, 26, 4, 18]. However, none of them focuses on distributed queries over distributed resources, although the work in [18] is closer in spirit to this chapter in the sense that it adopts the same describing framework. Static DQP is described in [51, 42], whereas the work in [31] discusses query processing issues in detail.

9.2 A Framework for Analysis of AdQP

AdQP can be deemed as the main means of self-optimization in query processing, and, as such, it relates to autonomic computing. According to the most commonly used autonomic framework, which is introduced in [41], at the conceptual level, autonomic managers consist of four parts, namely monitoring, analysis, planning and execution, whereas they interface with managed elements through sensors and effectors. In line with this decomposition, a systematic discussion about distributed AdQP distinguishes between monitoring, analysis, planning and execution. Note that these parts need not necessarily correspond to distinct implemented components at the physical level.

Monitoring involves the collection of measurements produced by the sensors. In the context of query processing, the types of measurements include data statistics (e.g., cardinalities of intermediate results), operator characteristics (e.g., selectivities) and resource properties (e.g., machine CPU load). The feedback collected is processed during the analysis phase with a view to diagnosing whether there is an issue with the current execution plan. If this is the case, then an adaptation is planned, which can be thought of as an additional query plan along with operations that ensure final result correctness. Execution is concerned with the actuation of the planned adaptations. Planned adaptations are executed either immediately in simple scenarios, or, in more complex cases (e.g., when internal state of some operators must be modified first), after certain procedures have been followed.

In this chapter, we follow the approach in [18] and we provide a summary of the measurements collected, and the analysis, planning and actuation procedures that are encapsulated in each of the main AdQP techniques presented. Note that these aspects of AdQP may be arbitrarily interleaved with query processing. For example, in some techniques, measurements' collection occurs after query processing has been suspended (e.g., due to materialization points), whereas other techniques continuously generate monitoring information during query execution. Also, analysis and planning may be tightly connected, since, in some cases the analysis of the feedback is done in a way that identifies better execution plans as well. Due to this fact, we prefer to examine analysis along with planning. Note that other variants of this framework, such as the one in [27], may regard planning and execution as a single response phase, whereas, during monitoring, preliminary analysis may be performed to filter uninteresting feedback.

9.3 AdQP in Centralized Settings

The role of this section is twofold. Firstly, it provides a short review of the main techniques employed in centralized AdQP, which is thoroughly investigated in surveys such as the one in [18]. Secondly, it discusses the feasibility of applying such techniques in distributed settings.

9.3.1 Overview of Techniques

In broad terms, the objective of conventional AdQP is to take actions in light of new information becoming available during query execution in order to achieve better query response time or more efficient CPU utilization. Although AdQP can be applied to plans consisting of any type of operators, there exist operators that naturally lend themselves to adaptivity in the sense that they facilitate plan changes at runtime. Such operators include symmetric hash joins and the proposals that build on top of them (e.g. XJoins [63]), multi-way pipelined joins (e.g., [65]) and eddies [2]. All the operators mentioned above can be complemented with additional operators that encapsulate autonomic aspects within their design; for example, certain operator implementations provide built-in support to adapt to the amount of the memory available (e.g., [52]).

Eddies constitute one of the most radical adaptive techniques on the grounds that they do not require explicit decisions on the ordering of commutative and associative operators (e.g., selections and joins). This results in a much simpler query optimization phase. Eddies have been proposed in order to enable fine-grained adaptivity capabilities during query execution; actually, they allow each tuple to follow a different route through the operators. More specifically, in eddies, the order of commutative and associative operators is not fixed and adaptations are performed by simply changing the routing order. To this end, several routing policies have been

proposed. The eddy operator is responsible for taking the routing decisions according to the policy adopted and monitoring information produced by the execution of tuples. As an example, assume that a long-running query over three relations of equal size is processed with the help of two binary joins. At the beginning, the selectivity of the first join is much lower than the selectivity of the other and the eddy will route most of the tuples to the most selective one. Later, the second join becomes more selective (e.g., due to the existence of a time-dependent join attribute); the eddy will be capable of swapping the order of join execution. Any static optimization decision on the join ordering would fail to construct a good plan in such a scenario where a different ordering yields better results in different time periods during query execution.

Hybrid approaches that combine eddies with more traditional optimization have been proposed as well. For example, in [49] and [48], a methodology is proposed where multiple plans exist and the incoming tuples are routed to these plans with the help of an eddy. Each such plan is designed for a particular subset of data with distinct statistical properties. In general, several extensions to the original eddy operator have been made (e.g., [56], [16], [10], [14]).

Another notable centralized AdQP technique has been proposed in [6], which adaptively reorders filtering operators. This proposal takes into account the correlation of predicates and can be used to enhance eddies routing policies. It has also been extended to join queries [7]. In general, join queries are treated in a different manner depending on whether they are fully pipelined and whether adaptations impact on the state that is internally built within operators because of previous routing decisions. Non-pipelined join queries were among the first types of queries for which AdQP techniques have been proposed. Such techniques are typically based on the existence of materialization points and the insertion of checkpoints, where statistics are collected and the rest of the adaptivity loop phases may take place if significant deviations from the expected values are detected (e.g., [40]). Two specific types of generalizations of these works are referred to as progressive optimization [47] and proactive optimization [5], respectively. Adaptive routing history-dependent pipelined execution of join queries is one of the most challenging areas in AdQP, where proposals exist that either use conventional query plans (e.g., [39]) or eddies (e.g., [16]).

9.3.2 On Applying Conventional AdQP Techniques in Distributed Settings

Undoubtedly, distributed AdQP techniques can benefit from the adaptive techniques proposed for a centralized environment. In DQP, each participating site receives a sub-query, which can be executed in an adaptive manner with the help of the techniques described previously. However, these adaptations, which are restricted to sub-queries only, are not related to each other, and, as such, they are not guaranteed to improve the global efficiency of the execution. For example, suppose that a set of operators in a query plan are sent to multiple machines simultaneously according

to the partitioned parallelism paradigm [19]. The execution of a query operator in a plan may benefit from partitioned parallelism when this operator is instantiated several times across different machines with each instance processing a distinct data partition. An eddy running on each of those machines could be very effective in detecting the most beneficial operator order at runtime; nevertheless, nothing can be done if the workload allocated to each of these machines is not proportional to their actual capacity.

In general, when AdQP techniques that were originally proposed for single-node queries are applied to full DQP, their efficiency is expected to degrade significantly due to the following reasons.

- Firstly, adaptations may impact on the state built within operators, as explained in [16, 73]. State movements in DQP incur non-negligible cost due to data transmission over the network. If this cost is not taken into account during the planning phase, then, the associated overhead may outweigh any benefits. Centralized AdQP techniques that manipulate the operator state in order to improve performance do not consider such costs, whereas, if state movement is avoided, then the adaptivity effects may be limited [16]. This situation calls for new AdQP techniques tailored to distributed settings.
- Secondly, several of the AdQP techniques mentioned above involve a final stitch-up (or clean-up) phase, which is essential for result correctness (e.g., [39]). As with state movement, when such a phase is applied to distributed plans, then additional overhead is incurred, which needs to be carefully assessed before proceeding with adaptations.
- Thirdly, direct applications of centralized AdQP techniques result in techniques in which there is a single adaptivity controller responsible for all the adaptivity issues. Obviously, this may become a bottleneck if the number of participating machines and/or the volume of the feedback collected is high. Scalable solutions may need to follow more decentralized approaches, which has not been examined in single-node settings.
- Finally, the optimization criteria may be different, since issues, such as load balancing, economic cost, energy efficiency are more likely to arise in DQP. These issues are closely related to load allocation across multiple machines, which is an aspect that does not exist in centralized environments.

Overall, the focus of distributed AdQP is different due to the fact that overhead and scalability issues are more involved, while load management is performed at a different level. The techniques described in the sequel address some of these issues.

9.4 AdQP for Distributed Settings: Extensions to Eddies

The original eddies implementation in [2] and its variants mentioned in Section 9.3 cannot be applied to a distributed setting in a straightforward manner. This is due to the fact that the eddy architecture is inherently centralized in the sense that all tuples

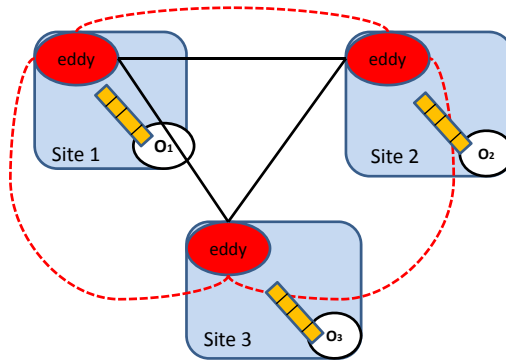


Fig. 9.1 Example of a distributed eddy architecture. The dashed lines show the statistics flow among the eddy operators, while the solid lines show the tuple flow.

must be returned to a central eddy; obviously, this paradigm leads to a single-point bottleneck, unnecessary network traffic and delays when operators are distributed. This section presents solutions to these problems.

9.4.1 Techniques

In [62], a distributed eddy architecture is proposed, which does not suffer from the limitations mentioned above. More specifically, in [62], each distributed operator is extended with eddy functionality. Moreover, a distributed eddy reaches routing decisions independently of any other eddies. Each operator places the received tuples in a first-come first-served queue. After a tuple has been processed, it is forwarded to the local eddy mechanism, which decides on the next operator that the tuple may be passed to based on the tuple's execution history and statistics. The operators in the distributed eddy framework learn statistics during execution and exchange such information among them periodically, e.g., after some units of time have passed or after having processed a specific amount of tuples. As in traditional eddies, routing decisions need not take place continuously; on the contrary, they may be applied to blocks of tuples in order to keep the associated overhead low [15]. Figure 9.1 shows an example of a distributed eddy architecture in a shared-nothing cluster of three sites.

The differences between centralized and distributed eddies are not only at the architectural level. The distributed eddies execution paradigm may be employed to minimize the result response time or to maximize the tuple throughput. For both objectives, it can be easily proved that the optimal policy consists of multiple execution plans that are active simultaneously, in the spirit of [17]; note that AdQP techniques typically consider the adaptation of a single execution plan that is active at each time point. However, analytical solutions to this problem are particularly expensive due to the combinatorial number of alternatives, and, in addition, they require the existence of perfect statistical knowledge; assuming the existence of perfect

Table 9.1 Adaptive control in Distributed Eddies [62].

Measurement: The eddy operators exchange statistics periodically regarding (i) the selectivity, (ii) the cost and (iii) the tuple queue length of the operators they are responsible for.

Analysis-Planning: The routing is revised periodically employing routing policies tailored to distributed settings. The routing decisions are made for groups of tuples.

Actuation: The planned decisions take effect immediately; no special treatment is needed since operators' internal state is not considered.

knowledge in a centralized statistics gathering component is not a realistic approach. So, the efficiency of distributed eddies relies on the routing policies. Interestingly, the most effective routing policies are different from those proposed in [2].

More specifically, several new routing policies are introduced in [62], in addition to those proposed for centralized eddies. Basic routing policies for centralized eddies include *back-pressure*, which considers operator load, and *lottery*, which favors the most selective operators. In the *selectivity cost* routing policy in [62], both the selectivity and the cost of the operators are considered in a combined manner. Although the above policy considers two criteria, it does not consider the queuing time spent while a tuple waits for processing in an operator's input queue. Thus tuples are routed to an operator regardless of its current load. In order to overcome this weakness, the *selectivity cost queue-length* policy takes into account the queue lengths of the operators as well. Contrary to the spirit of centralized eddies, the policies mentioned above are deterministic rather than probabilistic; note that this property is orthogonal to adaptivity. Finally, two more policies are proposed that route tuples in a probabilistic way that is proportional to the square of the metrics estimated by the selectivity cost and the selectivity cost queue-length policies, respectively. According to the experiments of the authors of [62], taking the square of the metrics exhibited better performance than taking the metrics alone.

Overall, during the monitoring phase, the raw statistics that need to be collected from each operator include its average tuple queue length, its selectivity and its processing cost. The routing policies effectively plan any adaptations. The actuation cost incurred when an alternative routing is adopted is negligible; the tuples are simply routed according to the new paths. Cases where the adaptation overhead cost is non-negligible, e.g., where operators create and hold internal state, which is affected by adaptations, are not investigated. The high-level summary of distributed eddies in [62] is given in Table 9.1.

The work of Zhou *et al.* in [72] extends the distributed eddies architecture proposed in [62] with SteMs [56]. SteMs add extra opportunities for adaptivity, since, apart from operator ordering, they can also change the join algorithm implementation (e.g., index-based vs. hash join) and the data source access methods (e.g., table scan vs. indexed access based on an attribute's column) at runtime. The local eddy operators utilize the traditional back-pressure and the lottery-based routing

Table 9.2 Adaptive control in [72].

Measurement: The eddy operators exchange statistics periodically. The exchanged statistics include (i) the average tuple queue length of the operators and (ii) the number of output tuples that are generated by the operators for the number of input tuples supplied to them.

Analysis-Planning: The routing is revised periodically. The back-pressure and the lottery based routing policies are employed for batches of tuples.

Actuation: The planned decisions take effect immediately; no special treatment is needed since operators' internal state is not considered.

strategies proposed for centralized settings [2]. In a distributed setting, the former routing strategy is used to accommodate the network transmission speeds and site workload conditions, while the latter reflects the remote operators' selectivity. Note that, as in [62], statistics are exchanged among the remote eddies at periodic time intervals, while routing decisions are made for groups of tuples. The statistics required by an eddy operator in [72] include (i) the average tuple queue length of the operators, and (ii) the number of output tuples that are generated by the operators. As in [62], the overhead incurred when an alternative routing is enforced, is negligible (see Table 9.2). Finally, FREddies is a distributed eddies framework for query optimization over P2P networks [36], which shares the same spirit as [62, 72].

9.4.2 Summary

The proposals described above are an essential step towards the application of eddies in DQP. However, a common characteristic is that the techniques in this category tend to avoid costly adaptations that involve manipulation of operators' internal state in order to diminish the risk of causing performance regression. A side-effect of such a reserved policy is that further opportunities to improve the performance of AdQP may be missed, as shown by successful relevant examples of AdQP techniques in centralized settings (e.g., [16, 21]). Another observation is that more research is needed in order to understand what type of routing policies is more efficient in distributed settings, and what are the benefits of probabilistic versus deterministic routing and of routing policies that are closer in spirit to flow algorithms [17].

9.5 AdQP for Distributed Settings: Operator Load Management

Load management can be performed at several levels; at operator-level load management, the main unit of load is an operator instance. In intra-operator load management, the different operator instances correspond to the same logical operator,

which implies that partitioned parallelism is employed and adaptivity is concerned with only a part of the query plan. On the other hand, inter-operator load management deals with adaptations that consider the whole plan, and operator instances may correspond to different logical operators.

9.5.1 Intra-Operator Load Management

Horizontal partitioning is a common approach to scale operators in a shared-nothing cluster [19]. In horizontal partitioning, an operator is divided into multiple instances. Each such instance is placed on a different site and processes different subsets of the input data. The operators on the different sites can work in parallel. Thus the result of the operator is given by aggregating the partial results that were produced by the different operator instances. For example, the result of an equi-join operator $A \bowtie B$ is given by the union of the partial results $A_i \bowtie B_i$, $i = 1, \dots, P$, where P is the degree of parallelism, i.e., the number of operator instances that work in parallel on different data. A_i and B_i correspond to the subsets of data that are processed at the i -th site and are partitioned according to the join attribute.

9.5.1.1 Background

A straight-forward way to enable query plans to benefit from partitioned parallelism without modifying the operators, such as joins and aggregates, is through the insertion of *exchanges* [30]. The exchange operator is one of the most notable non-intrusive attempts to parallel operator evaluation. The operator does not modify or filter any tuples but aims to distribute tuples across different operator instances. The exchange operator is logically partitioned into two components that may be hosted on different sites. The consumer component resides at a consumer operator instance and waits for tuples coming from the upstream producer operator instances. The producer component encapsulates the routing logic: it is responsible for routing the tuples to the consumer operator instances. The most common routing policies are hash-based, value range-based and round-robin.

Figure 9.2(top) shows an example of the partitioned execution of the hash-join $A \bowtie B$ in a shared-nothing cluster of four sites using an exchange operator. The tuples from the left relation A are used to build the hash-table, while the tuples from B probe the hash-table. As the different operator instances in sites 1 and 2 work in parallel, the time needed to complete the evaluation of the join operator equals the time needed by the slowest operator instance. Consequently, load imbalances can degrade the overall query performance. Load balancing aims to minimize the overall query response time by “fairly” redistributing the processing load among the consumer sites. By fairly is meant that the amount of work to be done on each site must be proportional to the capabilities of the site. In a volatile environment, the capabilities of participating machines or the relative size of the probing partitions may change at runtime. However, modifications to the routing policy so that the partitioning reflects better the current conditions lead to incorrect results, unless

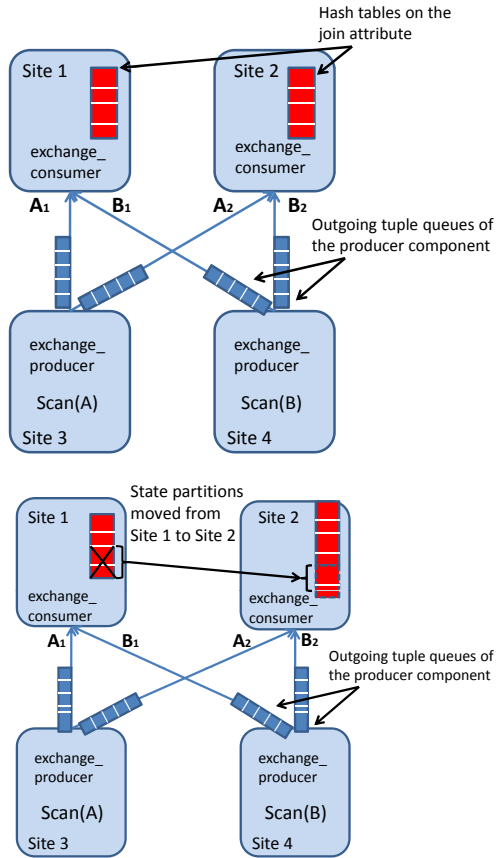


Fig. 9.2 Top: example of executing a hash-join over a shared-nothing cluster using the exchange operator. Bottom: example of state relocation.

these modifications are followed by the relocation of the corresponding buckets in the hash tables. This phenomenon is common to any partitioned operator that builds and maintains internal state during its execution. An example of state relocation is shown in Figure 9.2 (bottom), where parts of the hash-table in Site 1 are moved to the hash-table of Site 2 if the routing policy changes at runtime and more tuples are routed to Site 2.

9.5.1.2 The Flux Approach

Flux [59] is an operator that can be deemed as an extension to the proposals of the exchange and *river* [11] mechanisms, accounting for adaptive load balancing of stateful operators, such as windowed equi-joins and group-bys. Two policies are proposed for adaptive load balancing in clusters for settings with ample and

limited main memory, respectively. The first policy aims to transfer load (which entails state relocation as well) from an overloaded consumer operator instance to a less loaded consumer operator instance taking into account only the processing speed and idle times of consumers, while the second policy extends the former by considering memory management as well. The goal of the load balancing policy in a cluster with ample main memory is to maximize tuple throughput, through the minimization of utilization imbalances and the number of states moved. On the other hand, the constrained memory load-balancing policy tries to balance memory use across the cluster to avoid or postpone pushing states into disk.

In Flux, instead of having only one state partition per consumer instance, each consumer instance holds multiple “mini”-partitions. This is an effective mechanism for enabling fine-grained balancing [20]. In order to perform load balancing the following functionality is added. First, the consumer components maintain execution statistics tracked over monitoring periods. The maintained statistics differ with respect to the execution environment, i.e., whether the cluster has ample main memory or not. In the first case, the statistics are (i) the number of tuples processed per partition at the consumer side and (ii) the amount of time the consumer operator instance has spent idle, i.e., the amount of time the consumer component, which resides on the corresponding consumer operator instance, waits for input to arrive. From these statistics, the actual utilization of each node is derived. In the second case, i.e., when the aggregate main memory in the cluster is limited, the runtime statistics are (i) the available main memory at each consumer side, and (ii) the size of the partitions, along with indications of whether they are memory-resident or not. Second, the adaptations are coordinated by a global controller. The controller is responsible for collecting the runtime statistics from the consumer components and issuing movement decisions for load balancing.

In both policies mentioned above, load balancing is performed periodically and proceeds in rounds, where each round consists of two phases: a statistics collection phase and a state relocation phase. The duration of the state relocation phase impacts on the length of the next monitoring period with a view to avoiding scenarios where most of the time is spent shifting state partitions around. Also, in order to minimize the number of partition moves, for a given pair of sites, only one state partition is considered for movement, namely the one that reduces the utilization imbalance between the donor site and the receiver, provided that several threshold requirements are met.

The steps that take place when state partitions are relocated from one site to another are roughly the following: quiescing the partition to be moved, transferring the state partitions to the corresponding consumer sites and restarting the partition input stream. During quiescing, the consumer and the producer exchange messages in order to ensure that all in-flight tuples have been processed, and, as such, the consistency of the results is guaranteed. In addition, during the state movement period, each producer component marks the candidate state partition as stalled and buffers the incoming tuples for that state. After the state movement is performed successfully, all buffered tuples are redirected to the consumer operator instances that are selected to be the new hosts and the producer components resume directing tuples.

Table 9.3 Adaptive control in Flux [59].

Measurement: The following statistics are reported periodically from the local consumer components to the central controller: (i) the number of tuples processed per partition at the consumer side, (ii) the amount of time the consumer instance has spent idle, (iii) the available main memory at the consumer side, and (iv) the size of the partitions.

Analysis-Planning: Load balancing is coordinated by a central controller, which detects imbalances based on the measurements received. In case of imbalances, the controller forms pairs of sites that will exchange state partitions based on their utilization (in non memory-constrained clusters) or their excess memory capacity (in memory-constrained clusters) with the help of several thresholds.

Actuation: The steps are the following: (i) stall the input to the state partitions to be relocated, (ii) wait for in-flight tuples to arrive, (iii) transfer the state partitions to the corresponding consumer sites, and (iv) resume processing. The time spent for state movement will be used to define the next monitoring period length.

The policy for load balancing in a memory-constrained environment is similar to the one described above. However, in such an environment, state-movement is guided by the excess memory capacity criterion. The excess memory capacity at a consumer site is defined as the difference between the total memory size of the local states and the total available main memory. Similar to the previous policy, the state partitions selected to be moved are those that reduce the imbalance in excess capacity between pairs of sites. Flux can also perform secondary memory management, in the sense that each consumer site may autonomously decide to push and load state partitions into and from disks, respectively, in a round-robin fashion. The full details of the Flux adaptive load balancing approach can be found in [59]; a summary is presented in Table 9.3. Note that Flux can be complemented with fault tolerance capabilities [60].

9.5.1.3 Improvements on the Flux Approach

Paton *et al.* proposed some modifications to the original Flux operator in [54]. In one of the proposed variants, the execution proceeds as in the original Flux operator but state partitions are replicated instead of simply being moved. This entails higher memory requirements but, at the same time, manages to reduce the number of future state movements. In another variant, which assumes operators building hash tables, each hash-table bucket is randomly assigned to three sites. At hash-table build or probe phase, a tuple is sent to the two most lightly loaded of the three candidate sites that are associated with the bucket that the tuple is hashed to. During the probe phase, if a probe tuple matches a build tuple, the join algorithm generates a result from the probe occurred at the least loaded site, unless the matching (build) tuple is stored only on the other two sites. This variant reduces the adaptation overhead, which is mainly due to state movements, but incurs significant amounts of extra

Table 9.4 Adaptive control in [45].

Measurement: The following statistics are reported periodically from the local sites to the central controller: (i) that available main memory on the site, (ii) the size of partitions and (iii) the corresponding number of tuples that are generated from each partition.

Analysis-Planning: State relocation decisions are triggered by the central controller, when the available main memory imbalances exceed a user-defined threshold, based upon the productivity of each partition and the memory available.

Actuation: Similar to Flux [59] followed by a disk cleanup procedure in order to produce results from the disk resident state partitions.

work. Also, in [54], more advanced mechanisms for the actuation phase are investigated, which aim to cancel planned adaptations when the expected benefit does not outweigh the adaptation overhead.

Liu *et al.* have proposed techniques that deal with load balancing and secondary memory management of partitioned, stateful operators in an integrated manner [45]. As in Flux, the state relocation decisions are guided by a single controller, which periodically collects run-time statistics that are locally monitored at the remote sites during fixed-length time periods and triggers run-time adaptations. The criteria that guide the load balancing process are (i) the available main memory at each site, (ii) the partition sizes and (iii) the number of generated tuples from each partition.

State relocation is triggered by the controller when the available main memory imbalances among the sites exceed a user-defined threshold [45]. In that case, the most productive partitions from the site with the least available memory are moved to the site with the most available memory. Regarding secondary memory management, two different approaches can be followed. One local approach is to push the less productive partitions at each site into disk, when the amount of available memory is less than a user-defined threshold. Another global approach is to find the overall less productive partitions among all the sites and to push them into disk. The steps that take place during state relocation are similar to those in Flux with the addition of a disk cleanup procedure to produce results from the disk resident state partitions. The main characteristics of the integrated approach in [45] are summarized in Table 9.4.

The work in [32] extends Flux by supporting multiway windowed stream joins that are not necessarily equi-joins; moreover it focuses on the combination of load balancing and the so-called diffusion overhead. Load balancing is considered by allocating tuples to the less loaded machines. Diffusion overhead corresponds to tuple replications and intermediate join result transferring, which is needed to ensure correct result generation. Two algorithms are presented, which rely on partial tuple duplication. The first adaptively chooses a master stream, based on which the other streams are transferred, while the second builds upon a greedy solution of the weighted set cover problem [13]. The advantage of both approaches is that the routing is not based on the value of the tuples. Table 9.5 summarizes the main characteristics.

Table 9.5 Adaptive control in [32].

Measurement: The following statistics are reported periodically from the local processing sites and the aggregation site to the controller: (i) the usage and the capacity of the local CPU, memory and bandwidth resources (ii) the results throughput. The controller also monitors (iii) the input data streaming rates.

Analysis-Planning: The controller, apart from dynamically routing input tuples, dynamically selects the master stream and adapts its window segment length.

Actuation: The algorithm provides for special treatment of the intensionally duplicated tuples in order to ensure result correctness.

[66] addresses the same problem as in [32]. In [66], the notion of Pipelined State Partitioning (PSP) is introduced, where the operator states are partitioned into disjoint slices in the time domain, which are subsequently distributed across a cluster. Compared to [32], the approach in [66] does not duplicate any tuples and benefits from pipelined parallelism to a larger extent.

9.5.1.4 Summary

The previous discussion shows that, for the problem of intra-operator load balancing in DQP, several solutions with different functionality have been proposed. These solutions also differ in the trade-off between running overheads (which denote the unnecessary overheads when no adaptations are actually required) and actuation costs, which may accompany the execution of adaptivity decisions. The original Flux proposal is a typical example of an approach with low overhead but potentially high actuation cost, whereas other proposals in [54] mitigate the latter cost at the expense of higher overheads. Regarding the risk of causing performance regression due to costly adaptations, a limitation of the techniques mentioned above is that they do not consider the cost of moving operator state during the planning phase explicitly. The work in [29] fills this gap and revisits the problem of [59] by following a control-theoretical approach, which is capable of incorporating the overhead associated with each adaptation along with the cost of imbalance into the planning phase of the adaptivity loop. Initial results are shown to be promising, when machines experience periodic load variations. This is because the system does not move operator state eagerly, which is proven to be a more efficient approach [28].

9.5.2 Inter-Operator Load Management

While the techniques discussed in the previous section perform load balancing at intra-operator level, the approaches in this section perform load management at inter-operator level. In particular, two representative families of techniques that

correspond to different approaches to inter-operator load management are presented. The former, which is exemplified by [70], discusses cooperative load management with a view to achieving load balancing, whereas the latter, exemplified by [9], is concerned with a setting where each node acts both autonomously and selfishly. However, both groups of techniques deal with the problem of dynamically real-locating operators to alternative hosts without performing secondary memory management. Since no secondary memory management is done, the steps that take place during the operator relocation process are the following: (i) stalling the inputs of the operators to be moved and local buffering of the operators' input data, (ii) movement of the operators (along with the data inside their internal states) to their new locations and (iii) restarting of their execution. These steps constitute the typical operator migration procedure.

9.5.2.1 Cooperative Load Management

The work in [70], which is part of the Borealis project, assumes that data streams are processed on several sites, each of which holds some of the operators. Load balancing in [70] is treated as the problem not only of ensuring that the average load (e.g., CPU utilization) of each machine is roughly equal, but also of minimizing the load variance on each site and maximizing the load correlation among the processing sites. The rationale of the approach is described in the following example. Suppose that there exist load measurements of two operators hosted on the same site that have been taken during the last k monitoring periods; these measurements form two time-series. If both time series have a small correlation coefficient ([53]), then, allocating these operators on the same site is a good idea because it means that when one operator is relatively busy, the other is not. By putting these operators on the same site, the load variance of the host is minimized with a view to minimizing end-to-end latency. The average end-to-end latency degrades when highly correlated operators are co-located, since the operators may be simultaneously busy. In addition, if the load time series of two sites have a large correlation coefficient, then, their load levels are naturally balanced. Following the above rationale, the proposal in [70] tries to balance the load of a distributed environment by placing lowly correlated operators (in terms of load) at the same site, while maximizing the load correlation among the processing sites.

Under the proposed load balancing scheme, adaptations are performed periodically. The site and operator loads are locally monitored over fixed-length time periods and are reported to a single controller, which takes load balancing decisions. The load of an operator during a monitoring period is defined as the fraction of the CPU time needed by that operator in order to process incoming tuples (that arrived during that monitoring period) over the length of the monitoring period. The load of a site during this period is defined as the sum of the loads of all its hosted operators. Note that the controller keeps only the site and operator load statistics of the k most recent monitoring periods. Several algorithms are proposed in [70] for load balancing. However, all of them follow the same pattern: given a site pair, they decide which operators to transfer between the sites. The site pairs are decided by

Table 9.6 Adaptive control in Borealis [70].

Measurement: The sites periodically report the site/operator load statistics to the central controller. The controller retains the k most recent load statistics for each site/operator.

Analysis-Planning: The variance and the correlation is computed. Operator relocation is triggered periodically by the central controller. The operator redistribution algorithm first detects pairs of sites and then defines the operators to be moved.

Actuation: The re-distribution decisions are enforced through operator migration, which incurs non-negligible cost. This cost is only implicitly taken into account.

the controller based on their average load (the mean value of a site's load time series) following a procedure similar (to an extent) to the one used in Flux [59]. The proposed load balancing algorithms aim to optimize different objectives, i.e., the amount of load (operators) that is moved between a pair of nodes or the quality of the resulting operator mapping. The former technique considers operator migration cost implicitly. A summary of the adaptivity characteristics is given in Table 9.6.

In the context of the same project, a technique for failure recovery, which is equipped with dynamic load balancing characteristics has been proposed [37]. The proposed technique aims to provide low-latency recovery in case of a node's failure using multiple servers for collectively taking over the required actions. In particular the data that is produced and the data that is in the internal states of query fragments is backed up on a selected site. A site's failure is masked by other sites, which host backups and collectively rebuild the latest aggregate state of the failed server. However, new queries that may be submitted for execution or changes in the input streaming rates may change the sites' load and consequently the failure-recovery time. To solve this problem, the proposal in [37] may adaptively relocate the query fragment backups and move them from heavily loaded sites to less loaded ones.

Wang *et al.*, in [67], deal with a problem that is similar to the one in [70]. The distinct feature of this work is that operator placement decisions are based on mechanisms inspired by the physical world. In the physical world, each physical object tries to minimize its energy, whereas its behavior is driven by several types of potentials and the potential energy of an object depends on the location of other objects. In [67], each query operator is considered as a physical object, the potential energy of which reflects its output latency and depends on the site and on the network load conditions. The operator/site load is estimated as in [70], while the network load accounts for the overhead incurred by the network transmissions. As in [70], operator redistribution is performed by a single controller at periodic time intervals. All execution sites monitor the operator/site and network load conditions over fixed-length time periods and send the appropriate statistics to this controller. The controller performs load balancing utilizing heuristics that approximate the optimal solution. In order to minimize the overhead that is incurred during operator movement, only the most loaded operators are considered for redistribution. The fact that network

Table 9.7 Adaptive control in Medusa [9].

Measurement: Each site monitors its load.

Analysis-Planning:

The operator relocation process is triggered asynchronously when (i) a site becomes overloaded and (ii) another site (not-overloaded) is willing to accept (part of) its load in exchange of payment. The overloaded sites select a maximal set of operators that are more costly to process locally. Each not overloaded site, in turn, continuously accumulates load offers and periodically accepts subsets of offered operators.

Action: Typical operator migration, where operator migration overhead is considered small.

conditions are considered helps in mitigating the risk of performing non-beneficial adaptations, which is more likely to occur in [70].

9.5.2.2 Non-cooperative Load Management

The problem of load management in the Medusa project, which is a predecessor of Borealis, is treated under a different perspective, according to which the distributed systems are regarded as computational economies and the participants provide computational resources and accept to host and execute operators from other participants at a specified price [9]. Another difference with the load balancing techniques that are presented so far is that there is no single controller that decides which operators should be transferred to other hosts. In contrast, the hosts decide independently on the amount of load to transfer or accept.

In Medusa, the hosts aim to select an appropriate operator set in order to maximize the difference between the payment they receive and the cost incurred locally when processing this operator set. They negotiate with other hosts the amount of load to transfer or receive and the corresponding payment through contracts. The operator relocation process is not triggered at predefined time periods, but when, firstly, a site becomes overloaded and, secondly, another (not-overloaded) site is willing to accept at least part of the former's site load in exchange of a payment. To this end, the overloaded sites select a maximal set of operators that they are more costly to process locally than to offload, and offer them to another site. Each site continuously accumulates load offers and may periodically accept subsets of offered operators, on the grounds of higher unit-price offers. As such, the negotiation is asynchronous. If a new site accepts some of the offered operators, operator migration takes place. Note that in [9], a negotiation scheme is also proposed for non fixed-price contracts, due to the implications the fixed-price contracts may lead to. Table 9.7 summarizes the main characteristics of this adaptive load management approach.

9.5.2.3 Summary

This section discussed two different approaches to inter-operator load management. For cooperative scenarios, the solutions presented are interesting but still suffer from significant limitations, such as centralized control mechanism and increased risk to cause performance degradation due to the fact that the adaptation costs are not considered during planning explicitly. On the other hand, in non-cooperative scenarios an interesting alternative is proposed, according to which each node decides autonomously. Although the latter approach is more scalable, an issue that merits further investigation is to assess the messaging overhead in both approaches: in decentralized settings, nodes exchange messages in order to reach decisions as part of a negotiation protocol, whereas in centralized settings, nodes transmit monitoring information.

9.5.3 More Generic Solutions

Intra-operator and inter-operator load management techniques can be combined together. An example appears in [27], which considers partitioned pipelined queries running on distributed hosts. Intra-operator load management is responsible for balancing the load across partitioned operator instances in a way that reflects the runtime machine capacities. Inter-operator load management is responsible for detecting bottlenecks in the pipelines and removing them by increasing the degree of partitioned parallelism of the operators that form the bottlenecks. Another interesting feature of the same work is that operator state is not removed from any machine. Moreover, slow machines, for which the proportion of the workload assigned is decreased, do not participate in building operator state at the new sites. The responsibility for state movement rests with operators upstream in the query plan, which hold copies of data mainly for fault tolerance purposes at the expense of higher memory requirements.

A combination of inter- and intra-load management has been proposed for stream processing systems as well. This can be achieved through load sliding and splitting techniques, respectively [11]. Distributed eddies can be leveraged to behave in a similar way, too. However, understanding the interplay between efficient resource allocation and load balancing is a challenging topic because the goals are often conflicting, as explained also in [62].

9.6 AdQP for Distributed Settings: Other Techniques

In this section, we present techniques that deal with the problem of adaptive query optimization in distributed environments where the issues are not investigated at the operator level. In particular, we discuss proposals for adaptive parallelization of queries and web service (WS) calls (e.g., [64], [57], [68]). Additionally, a few

works that propose robust algorithms for distributed query optimization are briefly described (e.g., [34], [22]).

In [64], an adaptive technique is proposed that aims to optimize the execution of range queries in a distributed database. The tables are horizontally partitioned and the resulting partitions are replicated at multiple storage hosts. In order to minimize the query response time, the queries are executed in parallel. The hosts that store the data of interest are firstly identified, and then, the identified hosts process the same query using the local data partitions. However, setting the maximum level of parallelism does not necessarily minimize the result consumption rate, since, if a query is sent to too many storage hosts, results may be returned faster than the client who submitted the query can consume them. Apart from that, things become more complicated when multiple queries run in parallel and need to access the same storage hosts due to disk contention, which may slow down all queries. To solve this problem, Vigfusson *et al.* have proposed an algorithm that can adaptively (i) determine the optimal parallelism level for a single query and (ii) schedule queries to the storage hosts. In order to find the optimal parallelism level for each query, the algorithm randomly selects to modify the number of hosts that process the query in parallel for a short period. If this change results in an increase in the client consumption rate, then the change is adopted. The algorithm also employs a priority-based approach in order to schedule queries to hosts. The work in [57] deals with a similar problem, where adaptive approaches are explored for parallelizing calls to WSs. Also, in [38], substitution of data sources on the fly is supported to tackle data source failures.

Wu *et al.* proposed an adaptive distributed strategy for approximately answering aggregate queries in P2P networks [68]. In particular, data samples are distributed to sites for further processing. At each processing site, local aggregates are computed that are subsequently sent to a coordinator site, which combines them in order to produce the global aggregate value. The proposed strategy adaptively grows the number of processing nodes as the result accuracy increases with a view to minimizing the query response time.

Han *et al.* have proposed an extension to the initial proposal of progressive optimization in [47] to account for distributed environments [34]. This distributed proposal proceeds similarly to its centralized counterpart. Each plan fragment is marked at special points, where the optimality of the overall plan can be validated. The execution sites monitor the cardinalities of the local intermediate results at these special points and send a positive or negative vote for re-optimization to the controller; if the observed cardinality lies in the validity range, then the vote for re-optimization is negative, otherwise it is positive. The controller employs a voting scheme in order to decide whether re-optimization must be triggered or not. Several voting schemes are proposed in [34]. For example, in the *majority voting* scheme, re-optimization is triggered if at least half of the total execution sites vote for re-optimization. On the other hand, in the *maximum voting* scheme, re-optimization is triggered if at least one site sends a positive vote independently of the votes that the other sites send. Another extension to [47] is presented in [22]; the work in [22] focuses on queries that access data from remote data sources.

Finally, some earlier proposals defer resource allocation decisions until more accurate information about data statistics becomes available (e.g., [33, 71, 50]).

9.7 Conclusion and Open Issues

In this chapter, we investigated the state of the art in distributed adaptive query processing. The main techniques developed so far deal with issues such as extensions to eddies (e.g., [62]), intra-operator load balancing (e.g., [59]), inter-operator load balancing (e.g., [70]) and inter-operator load management with selfish hosts (e.g., [9]). These techniques differ significantly from their centralized counterparts, both in their objectives and in their focus. The objectives in distributed AdQP are more tailored to distributed settings, whereas more attention is paid to issues relating to the adaptivity cost, which is significant, especially when operators and data are moved over the network. Nevertheless, most of the techniques consider the increased adaptivity cost in an implicit heuristic-based manner, with the exception of the work in [29]. Apart from the adaptation costs, the overall performance of AdQP techniques needs to be investigated in a more systematic way, since only very few works are accompanied with theoretical guarantees about their behavior [6].

Other issues that have not been adequately addressed include scalability and the interplay between distinct AdQP techniques. Decentralized control in co-operative settings has been discussed in [62], but it is still an open issue how to apply the same approach in broader scenarios. Moreover, the relationship between load balancing and efficient resource allocation should be further explored. Also, in stream environments, load management may include load shedding techniques as well; it is worth conducting research to better understand the relationship between the AdQP techniques presented in this chapter and load shedding methodologies (e.g., [24]). For example, when the data production rate of a streaming data source increases beyond the capacity of the consuming operator, any technique from the following is applicable: to perform load shedding or to move the consumer to a more powerful node or to increase the degree of intra-operator parallelism of the consumer and subsequently perform load balancing. An interesting research issue is to develop hybrid techniques that combine these different approaches with a view to improving efficiency.

An additional interesting topic for further research is not merely to combine different query processing techniques, but also to combine AdQP with more generic adaptive techniques in distributed settings. For example, the problem of load balancing has also been studied in the area of P2P networks (e.g., [25, 55, 23]); it is not clear how AdQP behaves when applied to an adaptively managed distributed infrastructure. AdQP in distributed settings may also both benefit from and influence techniques in distributed workflow processing (e.g., [44]). Finally, advanced AdQP techniques should be coupled with techniques that mitigate the need for adaptivity, such as robust initial operator allocation (e.g., [69]).

References

1. Arpaci-Dusseau, R.H.: Run-time Adaptation in River. *ACM Trans. Comput. Syst.* 21(1), 36–86 (2003)
2. Avnur, R., Hellerstein, J.M.: Eddies: Continuously Adaptive Query Processing. *SIGMOD Record* 29(2), 261–272 (2000)
3. Babcock, B., Chaudhuri, S.: Towards a Robust Query Optimizer: A Principled and Practical Approach. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 119–130 (2005)
4. Babu, S., Bizarro, P.: Adaptive Query Processing in the Looking Glass. In: *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, pp. 238–249 (2005)
5. Babu, S., Bizarro, P., DeWitt, D.: Proactive Re-Optimization. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 107–118 (2005)
6. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive Ordering of Pipelined Stream Filters. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 407–418. ACM (2004)
7. Babu, S., Munagala, K., Widom, J., Motwani, R.: Adaptive Caching for Continuous Queries. In: *ICDE*, pp. 118–129 (2005)
8. Babu, S., Widom, J.: Continuous Queries over Data Streams. *SIGMOD Record* 30(3), 109–120 (2001)
9. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Contract-based Load Management in Federated Distributed Systems. In: *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 15–28 (2004)
10. Bizarro, P., Babu, S., DeWitt, D., Widom, J.: Content-based Routing: Different Plans for Different Data. In: *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pp. 757–768 (2005)
11. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., Zdonik, S.B.: Scalable Distributed Stream Processing. In: *CIDR* (2003)
12. Chu, F.C., Halpern, J.Y., Gehrke, J.: Least Expected Cost Query Optimization: What Can We Expect? In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 293–302. ACM (2002)
13. Chvatal, V.: A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research* 4(3), 233–235 (1979)
14. Claypool, K.T., Claypool, M.: Teddies: Trained Eddies for Reactive Stream Processing. In: Haritsa, J.R., Kotagiri, R., Pudi, V. (eds.) *DASFAA 2008*. LNCS, vol. 4947, pp. 220–234. Springer, Heidelberg (2008)
15. Deshpande, A.: An Initial Study of Overheads of Eddies. *SIGMOD Record* 33(1), 44–49 (2004)
16. Deshpande, A., Hellerstein, J.M.: Lifting the Burden of History from Adaptive Query Processing. In: *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pp. 948–959 (2004)
17. Deshpande, A., Hellerstein, L.: Flow Algorithms for Parallel Query Optimization. In: *ICDE*, pp. 754–763 (2008)
18. Deshpande, A., Ives, Z., Raman, V.: Adaptive Query Processing. *Foundations and Trends in Databases* 1(1), 1–140 (2007)
19. DeWitt, D., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM* 35(6), 85–98 (1992)

20. DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: Practical Skew Handling in Parallel Joins. In: Proceedings of the 18th International Conference on Very Large Data Bases (VLDB), pp. 27–40 (1992)
21. Eurviriyanukul, K., Paton, N.W., Fernandes, A.A.A., Lynden, S.J.: Adaptive Join Processing in Pipelined Plans. In: EDBT, pp. 183–194 (2010)
22. Ewen, S., Kache, H., Markl, V., Raman, V.: Progressive Query Optimization for Federated Queries. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 847–864. Springer, Heidelberg (2006)
23. Gedik, B., Liu, L.: PeerCQ: A Decentralized and Self-Configuring Peer-to-Peer Information Monitoring System. In: ICDCS, pp. 490–499 (2003)
24. Gedik, B., Wu, K.L., Yu, P.S., Liu, L.: GrubJoin: An Adaptive, Multi-Way, Windowed Stream Join with Time Correlation-Aware CPU Load Shedding. *IEEE Trans. Knowl. Data Eng.* 19(10), 1363–1380 (2007)
25. Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load Balancing in Dynamic Structured P2P Systems. In: Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), pp. 2253–2262 (2004)
26. Gounaris, A., Paton, N.W., Fernandes, A.A.A., Sakellariou, R.: Adaptive Query Processing: A Survey. In: Eaglestone, B., North, S., Poulouvassilis, A. (eds.) BNCOD 2002. LNCS, vol. 2405, pp. 11–25. Springer, Heidelberg (2002)
27. Gounaris, A., Smith, J., Paton, N.W., Sakellariou, R., Fernandes, A.A., Watson, P.: Adaptive Workload Allocation in Query Processing in Autonomous Heterogeneous Environments. *Distrib. Parallel Databases* 25(3), 125–164 (2009)
28. Gounaris, A., Yfoulis, C.A., Paton, N.W.: Efficient Load Balancing in Partitioned Queries Under Random Perturbations. *ACM Transactions on Autonomous and Adaptive Systems* (to appear)
29. Gounaris, A., Yfoulis, C.A., Paton, N.W.: An Efficient Load Balancing LQR Controller in Parallel Databases Queries Under Random Perturbations. In: 3rd IEEE Multi-conference on Systems and Control, MSC 2009 (2009)
30. Graefe, G.: Encapsulation of Parallelism in the Volcano Query Processing System. In: Garcia-Molina, H., Jagadish, H.V. (eds.) Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pp. 102–111 (1990)
31. Graefe, G.: Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25(2), 73–170 (1993)
32. Gu, X., Yu, P., Wang, H.: Adaptive Load Diffusion for Multiway Windowed Stream Joins. In: Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE), pp. 146–155 (2007)
33. Hameurlain, A., Morvan, F.: CPU and Incremental Memory Allocation in Dynamic Parallelization of SQL Queries. *Parallel Computing* 28(4), 525–556 (2002)
34. Han, W.S., Ng, J., Markl, V., Kache, H., Kandil, M.: Progressive Optimization in a Shared-Nothing Parallel Database. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 809–820 (2007)
35. Hellerstein, J.M., Franklin, M.J., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V., Shah, M.A.: Adaptive Query Processing: Technology in Evolution. *IEEE Data Eng. Bull.* 23(2), 7–18 (2000)
36. Huebsch, R., Jeffery, S.R.: FREddies: DHT-Based Adaptive Query Processing via Federated Eddies. Technical Report No. UCB/CSD-4-1339, University of California (2004)
37. Hwang, J.H., Xing, Y., Cetintemel, U., Zdonik, S.: A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In: Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE), pp. 176–185 (2007)

38. Ives, Z.: Efficient Query Processing for Data Integration. Ph.D. thesis. University of Washington (2002)
39. Ives, Z.G., Halevy, A.Y., Weld, D.S.: Adapting to Source Properties in Processing Data Integration Queries. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 395–406 (2004)
40. Kabra, N., DeWitt, D.J.: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 106–117. ACM Press (1998)
41. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. *IEEE Computer* 36(1), 41–50 (2003)
42. Kossmann, D.: The State of the Art in Distributed Query Processing. *ACM Computing Surveys (CSUR)* 32(4), 422–469 (2000)
43. Kossmann, D., Stocker, K.: Iterative Dynamic Programming: a New Class of Query Optimization Algorithms. *ACM Trans. Database Syst.* 25(1), 43–82 (2000)
44. Lee, K., Paton, N.W., Sakellariou, R., Deelman, E., Fernandes, A.A.A., Mehta, G.: Adaptive Workflow Processing and Execution in Pegasus. *Concurrency and Computation: Practice and Experience* 21(16), 1965–1981 (2009)
45. Liu, B., Jbantova, M., Rundensteiner, E.A.: Optimizing State-Intensive Non-Blocking Queries Using Run-time Adaptation. In: Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop (ICDEW), pp. 614–623 (2007)
46. Mackert, L.F., Lohman, G.M.: R* Optimizer Validation and Performance Evaluation for Distributed Queries. In: VLDB 1986 Twelfth International Conference on Very Large Data Bases, pp. 149–159 (1986)
47. Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., Cilimdžić, M.: Robust Query Processing through Progressive Optimization. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 659–670 (2004)
48. Nehme, R.V., Rundensteiner, E.A., Bertino, E.: Self-Tuning Query Mesh for Adaptive Multi-Route Query Processing. In: Proceedings of the 12th International Conference on Extending Database Technology (EDBT), pp. 803–814 (2009)
49. Nehme, R.V., Works, K.E., Rundensteiner, E.A., Bertino, E.: Query Mesh: Multi-Route Query Processing Technology. *Proceedings of the VLDB Endowment* 2(2) (2009)
50. Ozcan, F., Nural, S., Koksal, P., Evrendilek, C., Dogac, A.: Dynamic Query Optimization in Multidatabases. *IEEE Data Eng. Bull.* 20(3), 38–45 (1997)
51. Ozsu, M., Valduriez, P. (eds.): *Principles of Distributed Database Systems*, 2nd edn. Prentice-Hall (1999)
52. Pang, H., Carey, M.J., Livny, M.: Memory-Adaptive External Sorting. In: 19th International Conference on Very Large Data Bases, pp. 618–629 (1993)
53. Papoulis, A.: *Probability, Random Variables, and Stochastic Processes*, 3rd edn.
54. Paton, N.W., Buenabad-Chavez, J., Chen, M., Raman, V., Swart, G., Narang, I., Yellin, D.M., Fernandes, A.A.: Autonomic Query Parallelization using Non-Dedicated Computers: an Evaluation of Adaptivity Options. *The VLDB Journal* 18(1), 119–140 (2009)
55. Pitoura, T., Ntamos, N., Triantafyllou, P.: Replication, Load Balancing and Efficient Range Query Processing in dHTs. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) *EDBT 2006. LNCS*, vol. 3896, pp. 131–148. Springer, Heidelberg (2006)
56. Raman, V., Deshpande, A., Hellerstein, J.M.: Using State Modules for Adaptive Query Processing. In: Proceedings of the IEEE 19th International Conference on Data Engineering (ICDE), pp. 353–364 (2003)
57. Sabesan, M., Risch, T.: Adaptive Parallelization of Queries over Dependent Web Service Calls. In: Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE), pp. 1725–1732 (2009)

58. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access Path Selection in a Relational Database Management System. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data. ACM (1979)
59. Shah, M., Hellerstein, J., Chandrasekaran, S., Franklin, M.: Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In: Proceedings of the IEEE 19th International Conference on Data Engineering (ICDE), pp. 25–36 (2003)
60. Shah, M.A., Hellerstein, J.M., Brewer, E.A.: Highly-Available, Fault-Tolerant, Parallel Dataflows. In: Weikum, G., König, A.C., Deßloch, S. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, pp. 827–838. ACM (2004)
61. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: LEO - DB2's LEarning Optimizer. In: VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, pp. 19–28 (2001)
62. Tian, F., DeWitt, D.J.: Tuple Routing Strategies for Distributed Eddies. In: Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), pp. 333–344 (2003)
63. Urhan, T., Franklin, M.J.: XJoin: A Reactively-Scheduled Pipelined Join Operator. IEEE Data Engineering Bulletin 23(2), 27–33 (2000)
64. Vigfusson, Y., Silberstein, A., Cooper, B.F., Fonseca, R.: Adaptively Parallelizing Distributed Range Queries. Proceedings of the VLDB Endowment 2(1), 682–693 (2009)
65. Viglas, S.D., Naughton, J.F., Burger, J.: Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In: Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), pp. 285–296 (2003)
66. Wang, S., Rundensteiner, E.: Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-Slicing. In: Proceedings of the 12th International Conference on Extending Database Technology (EDBT), pp. 299–310 (2009)
67. Wang, W., Sharaf, M.A., Guo, S., Özsu, M.T.: Potential-driven Load Distribution for Distributed Data Stream Processing. In: Proceedings of the 2nd International Workshop on Scalable Stream Processing System (SSPS), pp. 13–22 (2008)
68. Wu, S., Jiang, S., Ooi, B.C., Tan, K.L.: Distributed Online Aggregations. Proceedings of the VLDB Endowment 2(1), 443–454 (2009)
69. Xing, Y., Hwang, J.H., Çetintemel, U., Zdonik, S.: Providing Resiliency to Load Variations in Distributed Stream Processing. In: Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB), pp. 775–786 (2006)
70. Xing, Y., Zdonik, S., Hwang, J.H.: Dynamic Load Distribution in the Borealis Stream Processor. In: Proceedings of the 21st International Conference on Data Engineering (ICDE), pp. 791–802 (2005)
71. Yu, M.J., Sheu, P.C.Y.: Adaptive Join Algorithms in Dynamic Distributed Databases. Distributed and Parallel Databases 5(1), 5–30 (1997)
72. Zhou, Y., Ooi, B.C., Tan, K.L.: Dynamic Load Management for Distributed Continuous Query Systems. In: Proceedings of the 21st International Conference on Data Engineering (ICDE), pp. 322–323 (2005)
73. Zhu, Y., Rundensteiner, E.A., Heineman, G.T.: Dynamic Plan Migration for Continuous Queries over Data Streams. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 431–442 (2004)

Chapter 10

Approximate Queries with Adaptive Processing

Barbara Catania and Giovanna Guerrini

Abstract. The traditional query processing approach, by which queries are executed exactly according to a query execution plan selected before query execution starts, breaks down in heterogeneous and dynamic processing environments that are becoming more and more common as query processing contexts. In such environments, queries are often relaxed and query processing is forced to be adaptive and approximate, either to cope with limited processing resources or with limited data knowledge and data heterogeneity. When approximation and adaptivity are applied in order to cope with limited processing resources, possibly sacrificing result quality, we refer to as Quality of Service (QoS)-oriented techniques. On the other hand, when they are a means to improve the quality of results, in presence of limited data knowledge and data heterogeneity, we refer to as Quality of Data (QoD)-oriented techniques. While both kinds of approximation techniques have been proposed, most adaptive solutions are QoS-oriented. In this chapter, we first survey both kinds of approximation and introduce adaptive query processing techniques; then, we show that techniques which apply a QoD-oriented approximation in a QoD-oriented adaptive way, though demonstrated potentially useful on some examples, are still largely neglected.

10.1 Introduction

The last decade has been characterized by a radical modification of query processing requirements. It has seen the raise of new applications with data querying needs and a substantial evolution of the processing environments. As discussed in Chapter 1, these emerging data intensive applications and novel processing environments, such as data integration applications, web services, data streams, P2P systems, and

Barbara Catania · Giovanna Guerrini
University of Genoa, Italy

e-mail: [barbara.catania,giovanna.guerrini}@unige.it](mailto:{barbara.catania,giovanna.guerrini}@unige.it)

hosting, to mention a few, are characterized by high heterogeneity, limited data knowledge, extremely high variability and unpredictability of data characteristics and dynamic processing conditions. The higher and higher resource sharing and the increasing interactivity in query processing make even those data properties that are traditionally conceived as static (such as relation cardinality and number of distinct values for an attribute) to be rarely known a priori and difficult to be estimated. This radical modification of requirements impacted in two main ways query processing. On one side, the need emerges to adapt the processing to dynamic conditions, thus giving up the a priori selection of a single execution strategy, fixed before processing starts. Techniques allowing to modify the execution strategy during the query execution are known as *adaptive* techniques. On the other side, *relaxed* queries are often executed, either as a user or a processor choice, or query evaluation is *approximate*, in order to cope with data heterogeneity, with limited data knowledge during query specification, or with limited available resources during processing.

More precisely, a query processing technique is said to adaptive if the way in which a query is executed is changed on the basis of the feedbacks obtained from the environment during evaluation. Specifically, we refer to: (i) as *subject* of the adaptive technique the elements in the processing affected by the adaptation, that are changed during the processing (e.g., the query execution plan, or the assignment of load to processors); (ii) as *target* of the adaptive technique what it attempts at adapting to, that is, the properties monitored and the feedbacks collected during evaluation (e.g., data characteristics, arrival rates, network condition, processors load); (iii) as *goal* or *aim* of the adaptive technique the parameter(s) appearing in the objective function, that is, what it attempts to maximize/minimize (e.g., result data quality, time, throughput, energy consumption).

A query is said to be relaxed if its result is either stretched or shrunk when the results of the original query are too few or too many. In this chapter, preference-based queries like top- k or skyline queries (see Chapters 2 and 3) are considered relaxed queries: they can be thought as a shrinking approach with respect to the overall set of possible results, since they reduce the cardinality of the ranked result dataset, or as a stretching approach with respect to the set of optimal results. A processing technique is said to be approximate if it does not produce the exact result but an approximate one, possibly with some guarantees on the “distance” of the generated solution from the exact one.

While approximate query processing (ApQP) for stored data is an option, for data streams it is always a must, due to the impossibility to cope with the overall dataset during query execution. Indeed, data streams are intrinsically unbounded. Therefore, it is not possible to provide a precise semantics for the operators that need to access all data items before generating the result, like join and aggregates (blocking operators). A first approximation level for data streams therefore consists in providing an approximate semantics to blocking operators. This is usually done by introducing the concept of *window*. A window is a mechanism to superimpose a region of definite cardinality over a stream whose cardinality is unknown. Using windows, even blocking operators can retain their semantics at the price of returning an approximate, and continuously updated, answer.

In the following, for the sake of simplicity but with abuse of terminology, we refer to both relaxed queries and queries executed in an approximate way as to *approximate queries* (or *approximate techniques* when focusing on the query specification and/or processing approach). Each approximate technique is characterized by: (i) a *subject*, representing the query processing task or the data to which approximation is applied (e.g., query specification, through rewriting or preferences, or processing algorithm); (ii) a *target*, representing the information used for the approximation (e.g., ranking function, set of relevant attributes, similarity function, pruning condition, used summary); (iii) a *goal* or *aim*, i.e., the parameter(s) appearing in the object function of the technique, that is, what it attempts to maximize/minimize (e.g., result data quality, time, throughput, energy consumption).

Based on their aim, approximate and adaptive techniques can be classified into two main groups. When they are finalized at improving the quality of result, either in terms of completeness or in terms of accuracy, we refer to as *Quality of Data (QoD)-oriented* techniques. By contrast, when they are used in order to cope with limited or constrained resource availability during query processing, we refer to as *Quality of Service (QoS)-oriented* techniques. Usually, a QoD/QoS aim implies the presence of a QoD/QoS target. For example, in order to maximize/minimize completeness or accuracy, QoD parameters have to be taken into account for adapting or approximating query specification and processing. Often, both QoD and QoS parameters are taken into account, in order to provide a good trade-off between resource usage and data quality.

While both QoS-oriented and QoD-oriented approximate techniques have been proposed, most adaptive solutions are QoS-oriented. QoS-oriented approximation is often applied in an adaptive way, that is, when targeted at achieving a QoS goal (related to load, throughput, memory, etc.), approximation is applied adapting to runtime conditions, possibly ensuring that certain QoD constraints are met or, less frequently, with a QoD-oriented goal (that is, minimizing the introduced inaccuracy). By contrast, very few approaches apply QoD-oriented approximation techniques in an adaptive way. While some adaptive QoD-oriented approximate techniques exist [58, 86], they address the problem of the efficient computation of a relaxed query. That is, they only consider QoS-based parameters (specifically, response time) with the aim of adapting processing so to maximize efficiency but maintaining the same approximation degree (which is inherently specified in the operation on processing).

In [29], we claimed instead that QoD-oriented adaptive approaches for QoD-oriented approximation techniques, called *QoD² techniques* in the following, may also help in getting the right compromise between precise and approximate computations. For example, this can be achieved by providing execution plans which interleave both precise and approximate evaluations in the most efficient way, dynamically taking decisions concerning when, how, and how much to approximate. Examples of *QoD² techniques* will be provided in Section 10.2. Unfortunately, as far as we know, no general solutions have been provided so far for the problem described above. Some analogies hold between the considered topic and quality-based

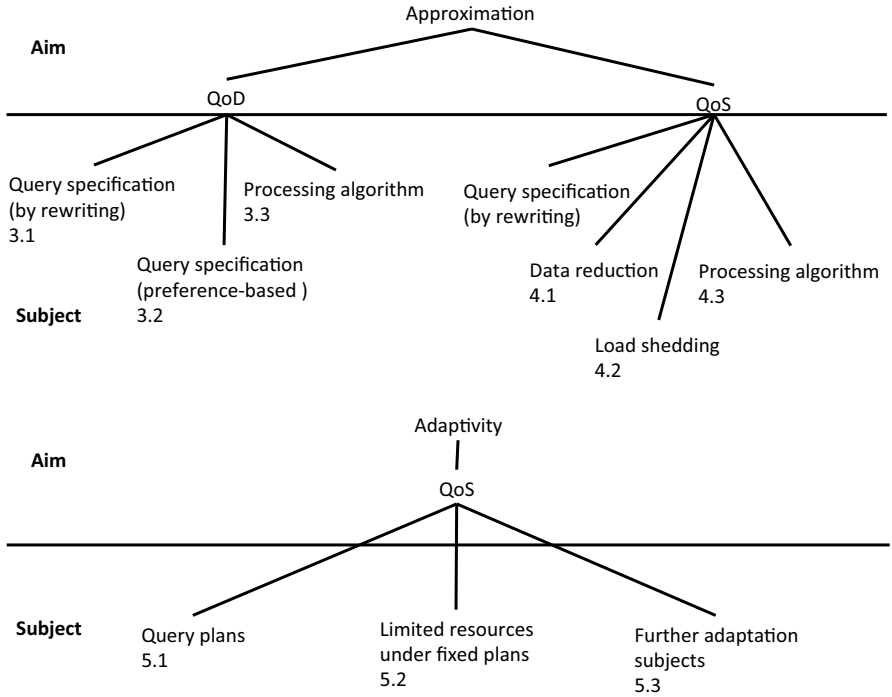


Fig. 10.1 Classification of the surveyed approximate and adaptive techniques.

query processors (see, e.g., [27, 126]), which take into account quality parameters in choosing the best execution plan. However, they usually consider a broad spectrum of quality criteria and have the goal of computing precise answers.

The aim of this chapter is twofold: (i) to present some relevant examples showing why and how QoD² techniques can be useful for advanced data management applications (Section 10.2); (ii) to survey currently existing approximate techniques and related adaptive solutions, with the aim of classifying and discussing existing approaches and pointing out their lack in providing QoD² techniques. To this purpose, approximation techniques will be first classified with respect to their aim type (either QoD-oriented - Section 10.3 - or QoS-oriented - Section 10.4 -) and then with respect to their subject; adaptive techniques will be classified with respect to their subject, since, as pointed out before, the aim of most of them is QoS-oriented (Section 10.5). Figure 10.1 shows how approximation and adaptive techniques will be classified, giving pointers to the related sections of the chapter. Section 10.6, based on the conducted analysis, focuses on when and how adaptivity is used in approximate query processing approaches, showing that QoD² techniques still constitute an open research direction.

10.2 QoD² Techniques: Some Examples

In this section we provide some concrete examples motivating and illustrating QoD² techniques.

10.2.1 *Adaptively Approximate Pipelined Joins*

Approximate joins are widely employed in data integration to cope with heterogeneous surface representation of the same real world object. The problem of record linkage [45] is at the heart of these data integration scenarios, where different and autonomously maintained tables are joined on the expectation that the values of some common attributes match, at least approximately. When two customer databases that belong to different organizations are merged, for example, it is reasonable to expect that the common customers can be found by means of a join. Unless those customers are represented in exactly the same way (i.e., the identifying attributes are the same, and with the same naming conventions) in both tables and in all instances, however, the result will in general be incomplete.

Existing techniques for off-line record linkage typically require advanced access to and analysis of the tables. This requirement is increasingly becoming unrealistic and the need for on-line techniques is emerging. The use of on-line techniques for data integration has been advocated also in [81] for data fusion. Off-line data integration is not applicable, for instance, in *mashup*-style integration scenarios, where two or more data sources are integrated on-the-fly, often by a third party who has no control over either source, or when the data to be joined are a continuous stream.

Consider, as an example, a mashup-based integration, where an organization collects data from various insurance companies into a large table of car accidents that have occurred nationwide over a period of time, and that is updated frequently. These data are then overlaid onto a map based on the accidents location, in order to visualize “accidents hot spots”. Suppose that the geographical information is obtained by joining this table—itsself collated from various sources—with a reference table containing an atlas of all streets in every city, along with their precise map location. In this case, the accuracy of the overlay map can be sacrificed in part, in return for a faster visual presentation.

In such dynamic, on-the-fly relational data integration settings, there is the need to reconcile values heterogeneity across sources, in order to ensure consistency and completeness of the integrated data. In this scenario, the use of exact joins to match records across sources may lead to incomplete integration, while approximate joins are computationally expensive. Approximate joins could be adopted with a *pessimistic* approach, that is, assuming that it is always worth paying the full computational cost of using a complex join operator, in return for the guarantee that all potential mismatches among tuples with sufficiently high similarity are caught.

In [77], the use of adaptive techniques for combining exact (fast) and approximate (accurate) joins when performing dynamic integration has been proposed. The adaptive algorithm uses an a-priori expectation of the join result size combined with the monitoring of join progress to statistically determine, at various points during query execution, which join operator should be used. Depending on its configuration, the algorithm can achieve various trade-offs between completeness of the join result and query execution time. The approach proposed is an *optimistic* one, whereby the initial assumption is that no mismatches will occur at all, thus a regular exact join is employed at the beginning. At the same time, however, a mechanism for detecting mismatches when they do occur, and for reacting to them by switching to a different query execution plan, where the exact join is replaced by a similarity join, is put in place.

The main issues that have to be solved in this context are: (i) how to assess that some mismatch is occurred and thus approximate join is needed; (ii) how to interleave approximate and exact join execution. As per the first issue, in the described scenario there is no prior knowledge of how many tuples in the source tables will fail to match when using an exact join. It may be the case, therefore, that for at least a portion of the tables a fast exact join could be used instead, without loss of joined tuples. In [77], the monitor component of the adaptive strategy is based on the assumption that a parent-child relationship is expected between the two input tables, a common case exemplified by the car accidents scenario. Under this assumption, join cardinality can be estimated easily and deviations from the expected cardinality reveal that mismatches are occurring and thus some approximation should be introduced.

For what concerns the second issue, approximate joins that can be elaborated in pipeline are needed. We recall that a pipelined plan executes all operators in the query plan in parallel, by sending the output data of an operator directly to the next operator in the pipeline, as opposed to materialized plans in which operators are applied in sequence, computing (and materializing if needed) whole intermediate results. An example of operator that can be elaborated in pipeline is Symmetric Set Hash Join, exploited in [77], which is a pipelined symmetric hash implementation of the q-gram based set join of [33]. Another example is a pipelined implementation of Trie-Join [123], that allows to assess tuple similarity as edit-distance constraints rather than as number of common q-grams. Both these operators can interplay with an exact symmetric hash join [125], provided that the needed data structures (the q-gram based hash tables and the trie, respectively) are built during exact join computation and that switches occur in quiescent states, as characterized by [46].

In this example, the target of both adaptation and approximation is QoD-oriented, while the aim of both adaptation and approximation is both QoD and QoS since the techniques aim at achieving the best trade-off between a QoD parameter, namely result completeness, and a QoS parameter, namely response time, by recurring to (more expensive) approximate joins when needed and by avoiding them if not needed.

10.2.2 Adaptive Processing of Skyline-Based Queries over Data Streams

Selection operations over streams are quite simple since they are not blocking operations. Therefore, each tuple can be evaluated as soon as it enters the system. However, users may not be acquainted of the actual data arriving in streaming, therefore they may issue queries that return an empty result set. As a consequence, users may want to modify the selection conditions and execute them again, until a satisfactory result is obtained. This may happen, for example, each time the query relies on constants whose relevance may vary over time.

Consider for example an application of habitat monitoring and assume that sensors have been located inside nests, returning several properties of the place around the nest, including light. Assume the user is interested in detecting the nests under a light above a certain threshold. Suppose also this monitoring should last for a long period, thus a continuous selection query is issued. Suppose the query is submitted during daytime, a given light threshold, say 400, is chosen, and the answer, probably non-empty, is computed. However, at sunset, the light is getting low and few (or no) data may be returned as answer. Two scenarios may arise: (i) this is exactly what the user wants and no modification to the query has to be specified; (ii) the user may anyway want some results to be returned, the closest to the specified conditions. In the second case, the system should modify the query in order to provide a non-empty result with accuracy guarantees. This behavior can be obtained, for example, by either changing selection conditions, similarly to what has been done in [68, 89] for stored data, or by relaxing the query using a skyline-based approach, similarly to what has been proposed in [72] for stored data and in [95] for sensor networks. As will be discussed further in Section 10.3.2, the basic idea of a skyline-based approach to query relaxation of selection and join operations is to use a relaxing function (usually, a numeric function) to quantify the distance of each tuple (pair of tuples) from the specified condition. The relaxed version of the query provides a non-empty answer while being ‘close’ to the original query formulated by the user.

While the empty answer problem has been deeply investigated for stored data, few proposals exist for data stream management (see Section 10.3). Specifically, no solutions have been proposed so far for skyline-based relaxation over data streams (while proposals for skyline query processing do exist - see, e.g., [83, 113, 115]). Even if such solutions were available, skyline-based computation is blocking and therefore a window-based computation would be required in order to restrict the set of items upon which dominance has to be checked. A trade-off therefore arises: the definition of skyline-based relaxation techniques may help in solving the empty answer problem but the price to pay, at least for selection, is the introduction of a window-based computation and therefore, in general, a decrease of performance.

A solution to this problem could be that of providing an approach for switching from precise selection operations to skyline-based ones as soon as, based on some dynamically monitored QoD parameters, the system understands that this is needed for improving result quality. The same technique may then switch from a

skyline-based computation to a precise one as soon as a QoD parameter indicates that a precise computation can again generate a result with QoD guarantee.

Similarly to the example presented in Section 10.2.1 in this scenario the target of both adaptation and approximation is QoD-oriented, while the aim of both adaptation and approximation is both QoD and QoS since the techniques aim at achieving the best trade-off between a QoD parameter, namely result completeness, and a QoS parameter, namely response time, by recurring to (more expensive) skyline-based computations when needed and by avoiding them if not needed.

Monitored QoD parameters should help in detecting an empty or few answers problem. Various alternatives could be devised and should be investigated. Of course, global cardinality constraints (e.g., return N items) are not suitable for a stream-based context. A possible alternative choice would be that of providing some rate constraints, specifying how many items should at least be retrieved from each W items processed or from each fixed period of time (e.g., return N items - or at least N items - every W tuples processed). Checking such constraints require a window-based computation that, for non blocking operations like selection, is not needed. Another simpler (and more efficient from a QoS point of view) option would be that of maintaining only the current selection selectivity, based on already processed tuples. Different parameters will obviously have different impact on query performance.

10.3 QoD-Oriented Approximate Queries

QoD-oriented approximate techniques provide approximate answers in situations where precise results are not always satisfactory from a user point of view. Indeed, due to high data heterogeneity and limited user knowledge about such data, precise evaluation may frequently produce result sets containing either empty/few answers or too many answers. A solution to these problems consists in modifying traditional queries, such as selections and joins, by relaxing their definition or by approximating their evaluation, in order to improve result quality, in terms of completeness and relevance with respect to the original query. Query rewriting (QR) and preference-based queries, such as top- k and skyline, are examples of QoD-oriented approximation techniques which relax the original query definition with the goal of returning a more satisfactory answer set. Of course, in defining QoD-oriented techniques, QoS guarantees have to be provided, in order to cope with the available resources in the most efficient way (a sort of second-level aim).

In this section, which aims by no means to be an exhaustive survey of the field, QoD-oriented approaches will be briefly surveyed, according to Figure 10.1 with respect to three subjects: query specification (by rewriting); query specification (by preferences); processing algorithm. While for stored data some techniques for each subject have been proposed, we are not aware of QoD-oriented proposals for data streams and related window-based queries based on QR. In presenting the approaches, we will point out the main properties and aspects upon which

Table 10.1 Some representative QoD-oriented approximate techniques.

Subject	Target	Approaches
Query rewriting	Values	[68, 89]
	Structure	[76, 131]
	Query predicates	[24]
Top- <i>k</i> queries	Ranking function	[41, 59, 80, 88, 91, 130]
Skyline queries	Set of relevant attributes	[26, 71, 83, 98, 106, 113, 114, 115]
System-specified top- <i>k</i> queries	Ranking function	[4]
System-specified skyline queries	Set of relevant attributes	[72, 82, 95]
Similarity-based join	Numeric similarity function	[109]
	Edit-based distance over strings	[33, 74]
	Token-based distance over strings	[53, 74]

approximation is applied (i.e., their target). The surveyed approaches concern relational as well as XML data. Some of the approaches mentioned for XML data are more extensively discussed in Chapter 6, to which we refer the interested reader for additional details. We remark that in this section the main approximation aim is always QoD-oriented, namely result data quality in terms of completeness or optimality with respect to some specified QoD criteria. Table 10.1 summarizes some representative approaches among the ones surveyed in the following in terms of approximation subject and target.

10.3.1 Query Rewriting

The aim of query rewriting is to rewrite the user query into a new one when the results of the original query are too few or too many. The main advantage of the query rewriting approach for query relaxation is that the generated queries can be executed using already existing query processing algorithms without the need of additional infrastructure. Of course, more efficient query processing algorithms can be provided in order to exploit the properties of the resulting query set.

Existing approaches, mainly proposed for stored data, can be classified into value-based, structure-based, and query-based depending on the information used for rewriting the query (i.e., depending on their target). More precisely: (i) value-based techniques rely on information concerning data distribution and query size estimation for rewriting queries, usually taking into account range and equality predicates on numerical and categorical attributes [68, 89]; (ii) structure-based techniques use schema or structure information (in case of semi-structured information like XML documents) during the relaxation process [76, 131]; (iii) query-based approaches relax queries based on properties of the employed query conditions. An example of query-based technique is given by the spatial relaxed topological selection operators and nearest neighbor operators proposed in [24] (see also Chapter 5).

10.3.2 Preference-Based Queries

In preference-based techniques, preferences are taken into account, as target, in order to generate the result, with the aim of providing best results first. In this category we include both top- k and skyline operators. Preferences can be either specified by the user or automatically chosen by the system. In both cases, they may correspond to ranking functions or sets of relevant attributes.

Preference-based queries have been proposed for both stored and stream-based data management. In the last case, they take into account that, based on the fact that data streams are unbound by definition, precise results are inherently not possible. Two main types of queries have therefore been considered: (i) continuous preference-based queries, whose result is continuously monitored over the stream, through the usage of a window, changing the result in a continuous way; (ii) one-spot preference-based queries, whose result has to be computed over the overall stream, assuming the set of possible items is fixed, in an approximate way with guarantees on errors. Typically, these last approaches rely on the usage of some sketch of data streams.

In the following, existing preference-based queries will be briefly surveyed by classifying them into three categories, namely: (user-specified) top- k queries, (user-specified) skyline queries, and system-specified preference-based queries.

Top- k queries. The aim of the top- k operator is to restrict the number of returned results to a fixed number (k), based on some ranking function (which constitutes the target of the relaxed query). Most of the existing top- k processing approaches rely on monotone ranking functions, which give the opportunity of optimizing top- k query processing, using some threshold value to prune the visit of data. However, for complex applications, the ranking function can be expressed as a generic numeric expression to be optimized [130]. A survey and a classification of existing top- k processing approaches for traditional and XML data has been proposed in [59]. Concerning XML documents, as discussed in Chapter 6, top- k approaches rely on ranking functions that take into account similarity of both the content and the structure of the documents with respect to the considered query [7, 55, 118].

When considering stream-based management systems, traditional top- k approaches have been extended to cope with the continuous query processing paradigm. As an example, in [91], a set of registered top- k queries are continuously evaluated over a sliding time-based or count-based window. Information about the ranking function is used to organize data in the window and continuously compute the result. On the other hand, in [41], a different buffer organization is provided, based on a dual representation of tuples by which any top- k query relying on a linear ranking function can be evaluated. Approximate solutions to precise top- k queries have been proposed in [88], with a space-bound guarantee. Some results presented in that paper have been proved to be imprecise in [80] and updated accordingly.

Skyline queries. The aim of the skyline operator is to return the best objects without relying on a specific ranking function, which sometimes could be cumbersome, but only specifying the attributes of interests. More precisely, given a set of points, each

representing a list of values for the list of relevant attributes, the skyline contains the points that are not dominated by any other point. A point A dominates a point B if it is better in at least one dimension and equal or better in all the others, by considering a specific ranking function [26]. Various algorithms have been proposed for skyline computation [26]. Index-based techniques (B-tree [26], bitmap [114], nearest neighbor [71, 98]) avoid scanning the overall set of data for skyline computation, improving performance with respect to basic techniques.

While top- k operators return a small result at the price of specifying a ranking function, which is not a simple task, skyline operators avoid this specification at the price of a larger result set, which, even for two dimensional interest attributes, may be quite huge. In order to address this problem, a common approach is to integrate both top- k and skyline advantages in a single technique. We refer the reader to Chapter 2 for further information about this topic.

Solutions for continuous skyline query processing have also been provided, both in centralized [115] and distributed [83, 113] architectures. Similarly to stored data, these approaches deal with totally ordered numerical domains. Other proposals exist to deal with skylines over categorical attributes over streams [106].

System-specified preference-based queries. Preferences are not necessarily specified by the user. Rather, they can be implicitly specified by the system and automatically taken into consideration during query execution. This is the case of the relaxation technique proposed in [72]. Here, relaxation is applied to relational selection and join conditions over numeric attributes by redefining the semantics of such operators based on a relaxing function, quantifying the distance of each tuple (pair of tuples) with respect to the specified condition, using a numeric function (usually, the difference between numeric values appearing in the condition and in the tuple(s)). The relaxed version of the query provides a non-empty answer while being ‘close’ to the original query formulated by the user, using a skyline-based approach. Another approach for automatic ranking database results has been presented in [4], where techniques for automatically deriving ranking functions are investigated, adapting typical Information Retrieval approaches to the database context.

When considering stream-based management systems, the only work we are aware of for what concerns system-based preferences has been proposed for sensor networks. Sensor networks are a special case of distributed stream management system where each data stream refers to data related to some environmental property collected through the usage of sensors. In such environments, an approach similar to that presented in [72] for relational data has been provided with the aim of solving the empty answer problem [95] (probing queries). The proposed solutions for processing probing queries take into account sensor network features and try to reduce communication costs and energy consumption during the execution. Another approach that can be classified as system-based is presented in [82], where various types of queries are executed in an approximate way at increasing accuracy levels. By refining the actual result set, the user can get further results at the price of a higher response-time.

10.3.3 Approximate Query Processing

Approximate query processing (ApQP) refers to all the techniques for executing a traditional query (e.g., a join) by using ad hoc query processing algorithms which automatically apply the minimum amount of relaxation based on the available data, in order to return a non-empty result more similar to the user request. Most QoD-oriented ApQP techniques concern the join operator [74] and face approximate match issues for strings [33, 53, 105] or numeric values [109]. These techniques are quite relevant in case of join between tables coming from different data sources. The presence of distinct strings representing the same information may arise for human factors (incorrect data entry or ambiguity during data specification), application factors (errors in database population or not enforced constraints), or obsolescence, since data are usually dynamic, that is, they are frequently updated and thus vary over time. Formally, an approximate (or similarity-based) join of two tables R_1 and R_2 is a subset of the Cartesian product of R_1 and R_2 . Specified attributes of R_1 and R_2 are matched and compared using a similarity function, instead of a usual equality predicate. The used similarity functions have strong analogies with those used in the context of data quality, for detecting that two values are distinct representations of the same real world entity (record linkage [73] or removal of duplicate records [45]).

Matching can be performed by considering as matching field either a single attribute, a set of attributes, or an entire tuple. The general problem thus becomes that of, given two field values, quantifying their similarity, as a number between 0 and 1. If the field is numeric, numeric methods can be used. If fields are strings, the problem is more challenging. The existing techniques can be broadly classified into edit-based functions, if they compare strings with respect to the single characters they contain [33, 74], and token-based, if they compare strings with respect to the tokens their contain, where a token is in general a substring satisfying specific properties [53, 74].

The naive method for executing an approximate join consists in computing the similarity score for each pair of fields and keeping only those whose similarity value is greater than a given threshold. This method is of course I/O and CPU intensive and therefore not scalable. Several algorithms have been proposed with the aim of reducing the number of pairs over which similarity is computed, by taking advantage of efficient relational join methods [74].

For XML data, ApQP has been deeply investigated, due to the very flexible structures and to the highly heterogeneous contexts in which XML data are used. The proposed approaches share the goal of integrating conditions over structure with the generation of approximate results. As discussed in Chapter 6, queries are typically expressed through twigs (i.e., small trees) to which data have to conform. Approaches differ on how conditions over structure are relaxed during approximation and on how similarity is quantified [6, 7, 104]. In addition, approaches for approximately joining XML trees [10, 54] and XML-specific record linkage techniques have been proposed [124].

10.4 QoS-Oriented Approximate Queries

The aim of QoS-oriented approximate techniques is to provide approximate answers, with accuracy guarantees, to computationally expensive operations also in environments characterized by limited or unavailable resources, where a precise result can be obtained only at the price of a unacceptably high response time, communication overhead, occupied space, or it cannot be obtained at all. QoS-oriented techniques have been mainly defined for queries to be executed over either huge amount of data (as in data warehousing systems and in stream-based management systems) or complex data (like spatial data) or because corresponding to very expensive computations (as multiway joins).

In QoS-oriented approximate techniques, the parameters with respect to which efficiency is evaluated (i.e., the specific QoS goal) depend on the specific environment to be considered. Specifically, while processing time and occupied space are always relevant, communication cost is a parameter to take into account in distributed contexts. On the other hand, the target corresponds to the specific information taken into account during the approximation process.

Concerning the subject, three main aspects have been considered for approximation: query rewriting, data representation, processing algorithm. QoS-oriented approximation techniques based on rewriting have been called static techniques in [90] and have mainly been proposed for stream data management. Static techniques modify the queries as soon as they are submitted to the system with the aim of using less resources at execution time. This can be done by statically changing some query conditions or parameters that may have an impact on resource usage. This includes windows, since by reducing the window size both memory and CPU time can be reduced, and sample rate, for those languages allowing its specification [90].

In techniques based on the approximation of data, data themselves are approximated with the aim of reducing or simplifying the dataset over which queries have to be executed. This approach is used each time it is too expensive to deal with values of a given domain (as it happens for spatial data) or with the overall set of objects (as it happens in data warehousing applications or data streams). In the last case, in presence of data streams, the term *load shedding* is typically used to define the process of dropping excess load from the system, in order to process only a subset of input data. Due to the specificity of load shedding approaches, in the following we use the term *data reduction* to refer any QoS-oriented data approximation approach but load shedding and we will describe the two groups of techniques separately. Specific query processing algorithms are usually provided to deal with the new simplified dataset, obtained by data reduction. With respect to techniques for generating precise results over non-approximate data, algorithms designed for approximate data are usually more computationally efficient since the size and the structure of input data is simplified and, as a consequence, also the new developed algorithms are.

When the processing algorithm is approximate, the input dataset is not necessarily changed but the processing used to execute a query against it is modified in order

Table 10.2 Some representative QoS-oriented approximate techniques.

Subject	Target	Approaches
Data reduction	Index-based data structures	[12, 56, 132]
	Samples	[2, 14, 32, 66, 108]
	Histograms	[42, 62, 99]
	Wavelets	[30, 122]
	TuG synopses	[11]
	Sketches	[8, 22, 36, 37, 38, 85, 103, 120, 128]
Load shedding	Random drop/ reducing probes	[69, 116]
	Semantic drop/discarding tuples	[15, 40, 75, 116, 117]
	Semantic drop/reducing probes	[47, 48, 49, 57, 112]
	Synopsis usage	[101]
Approximate processing algorithm	Pruning conditions	[5, 9, 35, 39, 129]
	Heuristics	[51, 60, 70, 96, 97]

to generate an approximate result in an efficient way, with respect to the available resources. Designing good approximate algorithms is a very complex task since they have to fully exploit the inherent properties of the data while making an efficient usage of available resources.

In the following, QoS-oriented approaches will be briefly surveyed with respect to their reference subject. In presenting the approaches, we will point out the main properties and aspects upon which approximation is applied (i.e., their target) and the specific goals. Table 10.2 summarizes some representative approaches among the ones surveyed in the following in terms of approximation subject and target.¹

10.4.1 Data Reduction

Data reduction refers to all techniques in which input data is summarized into a synopsis, significantly smaller than the input dataset, for which ad hoc efficient query processing algorithms can be defined. Historically, summaries have been used for selectivity estimation, recently they have been proved to be very useful for approximating query answers. On stored data, summaries are often pre-computed for the overall dataset and used mainly to provide computationally efficient approximate algorithms for expensive operations, e.g., join and aggregates, in environment characterized by massive data sets, like data warehouses. In general, there is a trade-off between the size of the summary and the accuracy of the result. Summaries can also be used, especially for window-based operations, for returning approximate results in an efficient way or for allowing computations over past data, in data streams. More precisely, for operations that need to access past data (like aggregates), data reduction is often a need. For other operations (like join), data reduction is an option to further improve efficiency, in presence of limited resources.

As pointed out in [50], when using summaries for query processing, the main challenges are: (i) what summary to maintain in a limited space in order to

¹ Notice that query rewriting is not further discussed in the following.

maximize accuracy and confidence of the approximate result that can be computed starting from it; (ii) how the summary can be efficiently updated as soon as data it refers change. Of course, different types of summaries are tailored to the execution of different operations. For example, table-level summaries, such as histograms or wavelets, are usually not suitable for capturing join-based correlations since their aim is to summarize the content of a single table. In this case, schema-level synopses (such as join synopses introduced in [3]) can be used to summarize the combined join and value distribution across several tables.

Various types of summaries have been proposed. For a survey concerning summaries for stored data, we refer the reader to [23, 50]. In the following, we briefly present some of the more relevant proposals.

Index-based data structures. The usage of index-based data structures as summary is quite useful each time processing domain values have an intrinsically high computational cost while dealing with key values is computationally less expensive. This is what happens for spatial data (see Chapter 5). The introduction of an approximate representation of the exact geometries is a well-known approach, typically used to filter out non-interesting data in the processing of spatial queries. A refinement step then returns the precise result, identifying all false hits generated by the filter step, at the price of executing potentially expensive spatial operations over the precise geometry. A more efficient but approximate solution to the approach described above is to consider the result generated by the filtering step as an “approximate query result”. Various approximate data representations can be used, with different accuracy levels (e.g., R-trees and their variants, using the Minimum Bounding Rectangle (MBR) as an approximation of data geometry [56], 4-color raster signature, 4CRS [12, 132]).

Samples. A sample is a small random subset of the dataset. Samples have been used in query processing for a very long time. They guarantee accurate estimates for aggregate operations (see e.g., [2, 32]), however their application to join operations is limited due to the fact that the join of two samples usually results in a non-uniform sample of the join result. Additionally, their effectiveness degrades when the underlying data distribution is skewed. Join synopses (i.e., samples of the join result) have been proposed to cope with this problem, but they can be used only for foreign-key joins, known beforehand [3].

Histograms. Histograms approximate data in one or more attributes by grouping values into buckets and approximating attribute values and their frequencies in the dataset based on statistics maintained for each bucket. Nowadays they represent a relevant approach for ApQP for both aggregate and non-aggregate operations [62, 99]. The main problem related to histograms is their construction cost and storage overhead, which make them not usable for high dimensional data.

Wavelets. Wavelets are a mathematical tool for the hierarchical decomposition of functions. Few wavelet coefficients can be used as a synopsis for handling aggregate [30, 122] and non-aggregate queries; such queries can be executed directly over the wavelet-coefficient synopsis to provide accurate answers in an efficient way [30].

TuG synopsis. Tuple Graph (TuG) synopses have been recently introduced as a form of schema-level summary that rely on graph-based models in order to summarize the combined distribution of joins and values in a relational database [111]. They guarantee accurate approximate answers for a large class of practical join queries, with aggregates and with several selection conditions over different tables.

Summaries in data stream query processing can be used for two different purposes: for approximating data inside a window or for approximating past data, required for example for general computations. While the various types of summaries introduced above for stored data can still be used (see, e.g., [14, 66, 108] for the usage of random samples, [42], for histogram-based approximation for window-based operations, [30] for wavelets), motivated by the fact that data in those environments is not totally available and therefore summaries should be incrementally computed in an efficient way, specific types of summaries, called *sketches* have been proposed.

Sketches are a specific type of summary, which can be incrementally computed. In order to be useful, sketches should be significantly smaller with respect to data they represent (typically, for window-based operations, polylogarithmic in the size of the data within the window) and should be updated very fast. Linear sketches are a special case of sketches that can be thought as a linear transform of the input: a sketch matrix multiplies the input, seen as a vector or matrix, and returns the sketch vector. As soon as a new data item comes, in order to update the sketch it is sufficient to multiply the new item for the sketch matrix (which often is implicitly defined). Sketches have been successfully used for queries with expensive filters [93], frequency-based queries [85], distinct-value queries [22], aggregates [8, 36, 38, 37, 42, 66, 110, 120, 128], spatial window queries as well as value range queries [82], and in combination with other summaries (e.g., with samples [103]). Specific types of constraints, defined for data streams, can be used to adapt the size of the sketch assuming that some data properties hold [20].

Features of specific approximation techniques for data streams based on summaries depend on the QoS parameters taken into account, i.e., on the specific aim which in turn depends on the considered architecture. For centralized stream management, memory limitations are typically considered [8, 22, 38, 42, 66, 85, 103, 120, 128]. On the other hand, communication costs [38, 120] are usually taken into account in distributed stream management systems. In sensor networks, since data is usually collected at some sensor nodes, working under energy limitations, summary-based approaches have been provided with the aim of reducing energy consumption [82].

A sensor network can be viewed as a distributed sensor data management system where the computations is pushed to the sensor nodes. Thus, data streams no longer consist of raw sensor readings but also of partial query results (in-network computation) [110] and summary-based approaches have therefore been adapted to this new architecture (see, e.g., [36, 37, 108]). Additionally, based on the consideration that sensor data refer to environmental features, it has been proved that, besides summary-based approaches, models of real-world processes can help to provide more robust interpretations of sensor readings: for example, they can account for

biases in spatial sampling, can help to identify sensors that are providing faulty data, and can extrapolate the values of missing sensors or sensor readings at geographic locations where sensors are no longer operational [43].

10.4.2 Load Shedding

Load shedding can be defined as the process of dropping excess load from the system. As a result, only a subset of input data is processed. There is therefore an analogy between load shedding and sampling, since both approaches reduce the number of tuples to be considered. However, while load shedding is applied on data queues, before entering query processing, sample is usually performed upon a set of tuples for which processing has already been activated. The target for load shedding approaches includes criteria by which the set of tuples to be shed are selected and drops are performed and can be either QoD- or QoS-oriented.

Load shedding approaches are typically proposed with the aim of optimizing the usage of limited CPU [15, 40, 47, 48, 49, 57, 69, 92, 101, 116, 117] or memory capabilities [40, 69, 75, 112], especially for window-based processing. Indeed, even with window predicates, operations may lack of CPU or memory resources in presence of high stream arrival rates. Typical operations considered for shedding are window-based join [40, 48, 49, 47, 57, 75, 69, 112, 116], multiway join [49], aggregates [15, 92, 117], select-project-join (SPJ) queries [101].

Drop operators can be either random, if they drop tuples based on the arrival rates but not on the properties of the tuple itself [69, 116], or semantic, also called utility-based, if they use a filter operator to decide which tuples have to be dropped, based on their utility in generating the result. To this aim, different metrics can be used, in order to: (i) minimizing the maximum error at outputs, by using either QoD or QoS utility functions that relate each output tuple to its utility [48, 49, 92, 116], learning approaches [47], or specific accuracy measures [15, 75]; (ii) producing the maximum subset result (also in case of random drops) [40, 57, 69, 117, 112]; (iii) producing a good sampled result [112].

The drop operation can be implemented in several ways. In general, it corresponds to simply discard input tuples from processing. In presence of windowed join, however, shedding can be realized by keeping the tuple in the window and limiting the number of probes, i.e., the number of checks against the opposite window for matching tuples [47, 48, 49, 57, 112]. Specific approaches depend on different assumptions about the tuples in a window: under the frequency-based approach [112], each join attribute value is assumed to have a roughly fixed frequency of occurrences in each stream (in [57], frequencies are dynamically maintained); on the other hand, the age-based or time-correlation approach [47, 48, 49, 112] assumes there is a time-based correlations between streams. An additional approach consists in removing tuples from the standard data flow but, instead of not further considering them, generating some synopsis of such tuples and use a fast but approximate shadow query plan to estimate the result [101].

Many load shedding operators are adaptive, meaning that their usage depends on dynamic processing conditions (see, e.g., [15, 47, 48, 49]).

10.4.3 Approximation of the Processing Algorithm

Several approximate algorithms have been proposed for stored datasets, to cope with high-dimensional data, typically arising in data warehousing or multimedia and spatial domains. They can be classified into one of the two following approximate strategies: search-space reduction and heuristic search techniques.

Search-space reduction techniques, instead of searching the overall data space for solutions, restrict the search-space by some means, thus reducing the computational cost of the processing but possibly affecting the accuracy of the result. Usually, search-space reduction techniques are obtained by altering some steps of existing algorithms for precise computation, by introducing weaker conditions (e.g., pruning condition could be weakened). Various techniques of this kind have been proposed for approximating kNN selection and join over spatial data (which become quite inefficient in high-dimensional spaces). They modify traditional kNN algorithms by reducing the search space through the usage of ad hoc pruning conditions and stopping criteria [9, 35, 39]. Even if top- k operators can be themselves considered as a QoD-oriented approximation approach, they can further be executed in an approximate way, using search-space reduction approaches, in order to improve performance. In those cases, approximate answers are associated with a probabilistic measure pointing out how far they are from the exact top- k answers [5, 129].

Heuristic search techniques might not find the best solution but they always find a good solution in reasonable time. Local search [96], simulated annealing [60, 70, 97], or genetic algorithms [51, 96] are usually applied in implementing heuristic-based approaches. They have been applied to both spatial join and kNN selection and join.

10.5 Adaptive Query Processing

In adaptive query processing (AdQP), the way in which a query is executed is changed on the basis of the feedbacks obtained from the environment during evaluation. The classical *plan-first execute-next* approach to query processing is replaced either by giving away the notion of query plan at all, as in routing based approaches, where each single data item can participate to the production of the final result taking its own way (*route*) through operators composing the query, or by a *dynamic optimization* process, in which queries are on-the-fly re-optimized through a two-steps solution. In the first step, the optimizer dynamically selects a new yet equivalent query plan based on system statistics gathered at runtime. In the second step, the system needs to be migrated from the query plan that is currently running to the plan identified in the first step (*dynamic plan migration*). A migration strategy must

guarantee that it will not alter the results produced by the system during as well as after the plan transition. This correctness requirement implies that results are neither missing nor contain duplicates. Both the approaches can be seen as instances of an adaptivity *measure-analyze-plan-actuate* loop [44], with different adaptation frequencies. As an extreme case, in routing based approaches adaptation is on a per-tuple basis. Note that though routing-based approaches do not rely on a notion of plan during execution, we can still talk of plans as a way to explain how data have been processed.

A query plan may consist of *stateless* as well as *stateful* operators. A stateless operator does not need to maintain intermediate data nor other auxiliary state information to be able to generate complete and correct results. A stateful operator, by contrast, at intermediate points of its execution stores data (or auxiliary information extracted from them) that have been processed so far in order to be able to generate complete and correct results. When processing queries over data streams, only pipelined plans can be exploited, and operators realizing joins and aggregates need to be stateful. Care must be taken to reduce the state continuous query accumulate.

Existing proposals can be classified according to different dimensions: (i) the frequency of the adaptivity loop (as in [63]); (ii) as plan-based, routing-based, or specifically targeted to continuous queries (as in [16]); (iii) the properties of the operation they apply to (as in [44]). In this section, which aims by no means to be an exhaustive survey of the field, we discuss some approaches broadly classified by adaptation subject, but pointing out as well the main properties and aspects to which query processing attempts at adapting (target), and which is the main adaptation objective (which we remark is a QoS objective in all the approaches). We notice that some of the described approaches [15, 20, 94, 116] introduce approximation, thus they have an impact on QoD, and some others [58, 86] process approximate operators (i.e., top- k), but the aim of the adaptation is QoS. Some adaptive load shedding approaches with a QoD objective exist (see Section 6), but they are not particularly relevant from an adaptive viewpoint and thus they are not discussed in this section.

We also remark that most adaptive techniques, even all those not introducing any approximation in the processing, do employ approximate data summaries, that are incrementally updated during processing, to collect data characteristics needed to drive adaptation.

Table 10.3 summarizes some representative approaches among the surveyed ones in terms of adaptation subject and target.

10.5.1 Adapting Query Plans

A first basic issue faced by AdQP techniques concerns adaptively coping with wrong cost estimates that lead to the selection of non-optimal plans. On stored data, this results from predicate selectivity estimates that may be revised according to runtime monitoring to reflect correlations between columns that were not considered by the cost model of the optimizers. On streaming data, data characteristics are unavailable

Table 10.3 Some representative AdQP techniques.

Subject	Target	Approaches
Query plan	data characteristics	[17, 67, 87, 86]
Tuple routing (post-mortem plan)	data characteristics	[25]
Tuple routing (post-mortem plan)	data characteristics and arrival rate	[11]
Query plan	data arrival rate and order	[65]
Query plan	data characteristics and arrival rate	[18, 102]
Query plan	processing environment	[58]
Operator scheduling	data arrival rates	[13]
Load shedding	data arrival rates	[15, 116]
Remote filter installation	data characteristics and arrival rate	[20, 94]
Load balancing	machine capabilities and actual workload distribution	[1, 52, 102]
Subquery sharing	workload, data and system conditions	[34, 84, 93]

and unreliable. More accurate and complete cost information may result from runtime selectivities, allowing either an appropriate data routing or to create a revised query plan reflecting them. Thus, the subject of adaptation (i.e., what is dynamically modified by adaptation) is the execution plan or the processing technique, and the adaptation goal is to minimize query processing time.

If we consider a distributed setting, other elements than wrong cost estimates affect the efficiency of query processing and thus have been coped with by adaptation. In case processing is made at one node, but data are gathered from different nodes, processing needs to adapt to network bandwidth and to rates at which data can be received from other nodes. The adaptation goal, in the distributed setting, in addition to minimize query completion time, is also continuity in producing results (result production rate, early production of first results). Dynamic adaptation in this context allows systems to react to (wide-area) network I/O delays and flow rates, due to different bandwidth or data provider resources, that can be subject to change without notice [64, 119, 121].

In query processing over data streams, both data characteristics and data arrival rates are varying and unpredictable. The cost of a plan for a continuous query depends on the current stream conditions (e.g., data distribution, arrival rate) and system conditions (e.g., query load, memory availability). Both these conditions may vary significantly over the lifetime of a long-running continuous query. The goal usually is to maximize query efficiency in terms of throughput.

In what follows, we briefly discuss the main proposals that face these issues by adapting the way the query is executed on data, distinguishing between routing and plan-based approaches, and, for the latter ones, classifying them according to the extent of partial work reuse they support upon plan migration.

Routing based approaches. Eddies [11] establish the order in which data are routed through operators dynamically on the basis of the costs and selectivities of the

operations, collected and monitored throughout query execution. This results in a per-tuple adaptation that does not consider at all the notion of query plan, rather it views query processing as routing of tuples through operators and realizes plan changes by changing the order tuples are routed. The eddy operator is a special operator that sits at the center of a tuple dataflow, intercepting the input and the output tuples of all other operators. It monitors the plan execution, and takes the decision concerning how to route tuples relying on some criteria grounded on the runtime information it collects. Most tuples exploit the route that is currently more efficient, while the rest explore other routes. Routes through operators simulates pipelined plans. After an adaptive routing-based technique has completed (that is, *post-mortem* [44]), we can explain what it did over time in terms of data partitions and corresponding adopted query plans.

The flexible routing approach of data through operators is particularly attractive in a data stream context and has been exploited in the TelegraphCQ [31] system. For join operators, however, it generally involves materializing partially processed tuples in hash tables (state modules) [100]. In such a way, the original eddies architecture is extended so that it can simulate multiway joins. Content-based routing [25] reverses the prevalent focus from adapting a single plan as data characteristics change, to detecting classes of data characteristics that can be used to route different data to different plans, through classifier attributes that allow to detect correlations and thus to produce conditional plans on-the-fly.

No reuse of partial results (Non-pipelined plans). In these approaches, most of which focus on join reordering during query evaluation [17, 67, 87], an optimizer is invoked at query runtime to identify a new plan on the basis of information on the processing progress to date. They are coarse grained approaches in that they either reuse complete join results or restart join evaluation from scratch, discarding results achieved with earlier plans.

Mid-query re-optimization [67] utilizes a run-time statistics collector and reconfigures only the unprocessed portion of the running query plan to improve performances. As in POP [87], the new plan can reuse materialized results from operators that are run to completion, but results of any partially computed subqueries are discarded, thus they are unsuitable for use with pipelined evaluation. Moreover, a plan may repeat work already performed by another plan. In Rio's proactive re-optimization [17] a *switch* operator is introduced that enables certain decisions on the plan to adopt to be deferred to run-time without discarding intermediate results. Switchable plans must have similar structures. Parametric query optimization [61] describes a query plan competing model to dynamically change the running query plan to another plan. The approach requires that before the query starts several plans have been chosen and are executed in parallel. After a while, the plan with the best exhibited performance thus far is kept running alone and the other ones are discarded. This allows for one-time dynamic plan migration.

The limited ability for dynamic plan migration makes these approaches not well suited for use in pipelined plans in which many operators are likely to be used simultaneously.

Pipelined plans and partial results reuse. Pipelined plans are addressed, and reuse of previously computed results supported, in [46, 65, 78]. In a distributed setting, Tukwila [65] proposes an approach of adaptive data partitioning that allows to discover and exploit order in source data, as well as data that can be effectively pre-aggregated. For instance, it combines hash join and merge join operators to take advantage of mostly ordered inputs. Tukwila works with pipelined plans, and supports reuse of previously computed results. Adaptation suspends a partially evaluated plan and creates a new plan that completes the work required to answer the query. To construct the query results from the values produced by the two plans, a post-processing in the form of a *stitch-up* phase is needed. This phase requires access to intermediate results produced by the two plans, and specifically to the state associated with a specific join algorithm (symmetric hash join).

In [78], adaptive reordering of pipelined indexed nested loop joins is considered. States are identified in which adaptation can safely take place. Adaptation however concerns only join order, whereas a single physical join operator, namely, index nested loop, is considered. Adaptive join re-optimization for pipelined plans based on the same approach but considering multiple join algorithms is investigated in [46]. Fine grained reuse of intermediate results is achieved, and redoing work is avoided, but some constraints are posed on the plans produced after re-optimization. Specifically, the new plan must complete partially evaluated scans and it makes use of intermediate data structures from the original plan.

In the data stream context, [28] (Aurora) adopts *pause-drain-resume* strategy for dynamic plan migration. That is, (i) the execution of the current query plan is paused, (ii) all existing tuples in the current query plan are drained out, (iii) the current plan is replaced by the new plan, and the execution restarted. This approach is appropriate to dynamically migrate a query plan that consists of only *stateless* operators, such as select and project. All intermediate tuples in a stateless query plan exist only in intermediate queues and can be cleaned completely by the drain step during the migration process. The pause-drain-resume strategy is unable to handle stateful operators such as window join with intermediate states. In the context of StreaMon, adaptive ordering of filter operators in pipelined plans over a single stream, to maximize throughput at all points in time, has been investigated in [18]. The challenge is to maintain the order in which the filters are evaluated over input stream tuples that is most efficient for the stream and system conditions at any point in time. The proposed A-Greedy algorithm devises the best filter ordering depending on current stream and filter characteristics, and can be applied to a wide class of multiway joins over streams. To minimize re-computation of intermediate results, an adaptive caching approach is proposed in [19]. The proposed A-Caching algorithm selects caches, monitors their costs and benefits in current conditions, allocating memory to caches and adapting as conditions change. Thus, adaptation is provided both to data characteristics and arrival rates, but also to memory availability. Like StreaMon [21], CAPE [102] supports adaptive processing at the level of operators, e.g., within a join operator, as well as at the level of query plans, e.g., switching among different plans for a query. Two different approaches for plan migration for stateful

operators are supported in CAPE, only for join re-ordering: moving states and parallel track. More flexible plan migration strategies are proposed in HybMig [127].

Finally, adaptive execution for top- k ranking queries is investigated in [58]. Different types of change in the optimality conditions of the current executing plan (e.g., cost parameters and fluctuations in the computing environment) are considered. Top- k ranking queries are indeed extremely relevant in less-stable environments, characterized by unexpected delays and frequent disconnections. The proposed algorithm allows the system to alter the current pipelined ranking plan at runtime and to resume with the new optimal (or better) execution strategy. The plan alteration mechanism employs an aggressive reuse of the old ranking state from the current plan in building the state of the new plan. The adaptive approach for top- k approximate XML query processing proposed in [86] and discussed in Chapter 6, by contrast, allows partial matches to the same query to follow different execution plans, so that a partial match that is highly likely to end up in the top- k set is processed in a prioritized manner, whereas a partial match unlikely to be in the top- k set follows the cheapest plan that enables its early pruning.

10.5.2 *Adaptively Coping with Limited Resources under Fixed Plans*

The arrival rate of a data stream may be extremely high or bursty, thus placing constraints on processing time or memory usage; typically, data must be processed on-the-fly as it arrives and can be spooled to disk only in background. A first issue that data stream processing systems have coped adaptively with is thus how to handle bursty data arrival rates. The main approaches are to drop unprocessed tuples to reduce system load (load shedding, see Section 10.4.2) and improving the efficiency of allocation of memory among query operators (operator scheduling). In both cases the goal is to improve system performances, in the first case, some approximation is introduced while in the second one the primary resource bottleneck that is aimed at overcoming is limited main memory. Approximation can be adaptively introduced in processing queries over data streams also to maintain a certain throughput or to reduce memory occupancy. This kind of adaptive approximation is typically driven by QoS metrics, ensuring that certain result precision guarantees are ensured.

Adaptive operator scheduling. When processing high-volume, bursty data streams, the natural way to cope with temporary burst of unusually high data arrival rates is to buffer the backlog of unprocessed tuples and process them during light load periods. However, it is crucial for the system to minimize the memory required for backlog buffering, which may otherwise exceed available memory during periods of heavy loads. An operator scheduling strategy, ideally invoked at the end of every unit of time (smallest duration for which operators should be run without preemption), selects an operator from those with non-empty input queues and schedules it for the next time unit. The operator scheduling strategy has a significant impact on the total amount of memory required for backlog buffering. The problem of how to

most efficiently schedule the execution of query operators to keep the total memory required for backlog buffering at a minimum, assuming query plans and operator memory allocation are fixed, is addressed in [13]. Since reasonable output latency must be ensured, the maximum runtime memory is minimized while maintaining the output latency within pre-specified bounds.

Adaptive load shedding. In case of bursty data arrival, the system drops input tuples to bring the system load down to manageable levels. As discussed in Section 10.4.2 alternative approaches are possible: a fraction of the input is dropped, and approximate answers are provided as output [15, 47], or entire windows are dropped so that the output is a subset of the actual one (subset result, [117]). Moreover, shedding may not simply discard data, it may also capture through synopsis properties of missing data [101]. In Aurora [116] shedding relies on QoS functions that specify the utility of processing each function from a data stream. A different quality of service function is provided for each query. In [15] load shedding is formulated as an optimization problem where the objective function is minimizing inaccuracy in query answers in the context of sliding window aggregate queries, taking also operator sharing into account. In [34, 40] the problem is addressed for windowed join between data streams, and the metric being optimized is the number of tuples produced in the approximate answer in [40], and the output rate of tuples in [34]. In [47] time correlation among windows is taken into account for window join (age-based load shedding). In [92] the problem is tackled for aggregates taking into account that each query has different processing cost, importance and maximum user-specified tolerated error.

Adapting filters at remote data sources. An approach is proposed in [94] to reduce communication overhead in a setting where distributed data sources stream data to a central processor. Filters are installed at remote data sources that adapt to changing conditions to minimize stream rates while guaranteeing that all continuous queries still receive the data necessary to provide answers of adequate precision at all times. Through an adaptive policy, the filter bound widths are adjusted continually to match current conditions. Many continuous queries with different precision constraints involving overlapping sets of data objects are considered.

Query plans for continuous queries often need to maintain significant runtime state in memory, which limits the number of queries the system is able to process simultaneously. For instance, a query plan for a windowed stream join query needs to store all tuples in the current window over the streams. The approach of [20] is to monitor input streams for various data or arrival patterns that can be exploited to reduce runtime state, without compromising correctness.

10.5.3 Further Adaptation Subjects

Further dimensions of adaptation include load balancing, that encompasses further issues in distributed query processing, and subexpression sharing, peculiar of continuous query processing (see also Chapter 9).

Adaptive load balancing. If the query is to be processed in a distributed way on several machines, dynamic adaptation extends to runtime load balancing by moving certain workload across machines. The issue is extremely relevant in environments where dynamic out-sourcing of processing tasks to non-dedicated, heterogeneous computers can take place and the effectiveness of parallelism depends more on the exploitation of the actual machine capabilities and efficient workload distribution, than on the way data is partitioned. Adaptivity in such environments mainly relates to runtime monitoring and learning the behavior of participating machines, the actual communication bandwidth between them, and the impact of this behavior on the progress of query processing. Moreover, it manifests itself as changes in the way available resources contribute to the query execution rather than as runtime modifications of the query plan, and the order in which data or operators are processed. For instance, in [52] a technique is proposed that dynamically balances intra-operator load across computational nodes both for stateful and stateless operations, and is capable of changing the degree of parallelism and moving load to new resources on-the-fly to overcome bottlenecks.

In distributed continuous query systems the basic unit being moved during the adaptation may be one whole operator, assuming that each operator is small enough to fit on one machine. We refer to this adaptation, supported for instance in Borealis [1] and CAPE [102], as *operator-level* adaptation. Many operators in continuous queries need states to keep tuples they have received so far for future processing. In case of high stream workloads, the states in one operator can grow too large to fit in the main memory of a single machine. Moreover, moving around large amounts of states at run time can be inefficient. Flux [107] addresses this problem by proposing strategies to divide one large operator state into many smaller partitions. One partition can then be treated as one moving unit during runtime adaptation. We refer to this type of adaptation as *state-level* adaptation. In [107] stateful operators with a single input (i.e., aggregate operators) are considered, whereas operators with multiple inputs and multiple states, such as binary or multi-way joins, which are more likely to have bigger operator states, are considered in D-CAPE [79].

Adaptive sharing of computation in continuous queries. The issue of minimizing redoing of work in multiple queries being continuously evaluated and sharing some subexpression is relevant not only for data streams but also for monitoring persistent data sets spread over a wide-area network (web sites over internet). NiagaraCQ [34] addresses adaptive regrouping of queries based on changes in workload, data or system conditions. No other dimensions of adaptation are however considered. An adaptive approach to shared filter evaluation is proposed in [93], where the goal is to minimize the overall cost of evaluation by sharing filter evaluations across multiple queries. They assume that operator costs and selectivities do not change over time, and different plans may be chosen for different tuples since the next filter to be evaluated is based on the results of filters evaluated so far. Sharing of computation among queries in eddies has been addressed in [84].

10.6 Conclusion and Discussion

Query processing in several new application contexts is characterized by two main features: adaptivity, in order to adapt the processing to dynamic conditions that prevent the selection of a single optimal execution strategy, and approximation, in order to cope with data heterogeneity, limited data knowledge during query specification, and limited resource availability, which make precise answers impossible to compute or unsatisfactory from a user point of view. The aim of the chapter was to survey currently existing ApQP and AdQP approaches and to show potential lacks in their combined usage. To this aim, based on the analysis performed in Sections 10.3, 10.4, and 10.5, we provide in Table 10.4 an overall picture of the query processing approaches both exhibiting an adaptive behavior and introducing some approximation, that we have separately discussed in the previous sections.

In Table 10.4, some representative approaches are classified with respect to the type of their target and aim. Notice that, since for approximation techniques a QoS/QoS aim usually implies the presence of a QoS/QoS target, a single dimension is considered. For each dimension, we highlight whether the approach relies on the usage of QoS or QoD parameters. For instance, the approach presented in [94] adaptively introduces some approximation, by omitting to transfer some tuples from remote data sources to the central processor (elaborating an aggregate operator). The introduced QoS-oriented approximation is adaptively achieved, the adaptive process is QoS-targeted at reducing communication overhead and has a QoS goal in minimizing the transfer rates of data from the sources to the central processor.

Table 10.4 Adaptively approximate query processing.

Adaptation		Approximation	
		QoD	QoS
Target	QoD		
	QoS	[58], [86]	[15], [20], [48], [94], [116]
Aim	QoD		[15], [48]
	QoS	[58], [86]	[15], [20], [48], [94], [116]

Table 10.4 highlights the following facts:

- Some approaches to QoS-oriented adaptive processing of QoD-oriented approximate queries have been proposed. Specifically, adaptive processing techniques for top- k queries on relational and XML data are proposed in [58] and [86], respectively, but the target and goal of the adaptation is QoS, namely processing efficiency.
- Adaptive approaches have been proposed for QoS-oriented approximation techniques (namely, load shedding and data summarization) in the data stream context [15, 20, 48, 94, 116]. Only few of them take QoD-information into account

as aim [15, 48]. Constraints on data (which are adaptively kept up-to-date) are exploited in [20] to improve QoS (specifically, to reduce memory overhead). However, none of the existing approach has QoD as a target of adaptation, consistently with the fact that none of them exploits QoD-oriented approximation.

From the performed analysis, it emerges that the definition of QoD-oriented adaptive approaches for QoD-oriented approximation techniques (i.e., QoD² techniques) has been so far neglected. However, the motivating and illustrating examples discussed in Section 10.2 that would fill the gap in the QoD-QoD cell of the table, show that QoD² techniques could be very relevant for various data management applications. Some preliminary efforts towards the definition of a reference framework for QoD² techniques have been presented in [29], where they are called ASAP (Approximate Search with Adaptive Processing) techniques, and some specific concrete techniques have already been proposed [77]. Much more work is required in order to determine their concrete impact on stored data and data stream management. As discussed in [29], we think it is important to start from the definition of a general framework, in order to capitalize on the experience gained in a decade of development of exact and QoS-oriented approximate adaptive technique. Adaptive techniques, indeed, have been proposed as largely independent solutions to similar problems being experienced in various data processing contexts. Only later they have been recognized as different instantiations of a same adaptive query processing approach, based on the common basic schema of adaptivity loop, with varying instantiation frequencies. As pointed out in existing surveys (e.g., [44]), adaptive query processing can take many forms, and, in general, a particular adaptive solution must find a particular point among the spectrum of trade-offs defined by a variety of different dimensions. In the development of QoD² techniques, reliance on that unified view will help to develop an organic set of techniques, coherently specified as different instantiations of a single framework.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryzkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The Design of the Borealis Stream Processing Engine. In: CIDR, pp. 277–289 (2005)
2. Acharya, S., Gibbons, P.B., Poosala, V.: Congressional Samples for Approximate Answering of Group-By Queries. In: SIGMOD Conference, pp. 487–498 (2000)
3. Acharya, S., Gibbons, P.B., Poosala, V., Ramaswamy, S.: Join Synopses for Approximate Query Answering. In: SIGMOD Conference, pp. 275–286 (1999)
4. Agrawal, S., Chaudhuri, S., Das, G., Gionis, A.: Automated Ranking of Database Query Results. In: CIDR (2003)
5. Amato, G., Rabitti, F., Savino, P., Zezula, P.: Region Proximity in Metric Spaces and its Use for Approximate Similarity Search. *ACM Trans. Inf. Syst.* 21(2), 192–227 (2003)
6. Amer-Yahia, S., Cho, S., Srivastava, D.: Tree Pattern Relaxation. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) *EDBT 2002*. LNCS, vol. 2287, pp. 496–513. Springer, Heidelberg (2002)

7. Amer-Yahia, S., Koudas, N., Marian, A., Srivastava, D., Toman, D.: Structure and Content Scoring for XML. In: VLDB, pp. 361–372 (2005)
8. Arasu, A., Manku, G.S.: Approximate Counts and Quantiles over Sliding Windows. In: PODS, pp. 286–296 (2004)
9. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. *J. ACM* 45(6), 891–923 (1998)
10. Augsten, N., Böhlen, M.H., Dyreson, C.E., Gamper, J.: Approximate Joins for Data-Centric XML. In: ICDE, pp. 814–823 (2008)
11. Avnur, R., Hellerstein, J.M.: Eddies: Continuously Adaptive Query Processing. In: SIGMOD Conference, pp. 261–272 (2000)
12. Azevedo, L.G., Zimbrão, G., de Souza, J.M.: Approximate Query Processing in Spatial Databases Using Raster Signatures. In: Advances in Geoinformatics, pp. 53–72 (2006)
13. Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D.: Operator Scheduling in Data Stream Systems. *VLDB J.* 13(4), 333–353 (2004)
14. Babcock, B., Datar, M., Motwani, R.: Sampling From a Moving Window over Streaming Data. In: SODA, pp. 633–634 (2002)
15. Babcock, B., Datar, M., Motwani, R.: Load Shedding for Aggregation Queries over Data Streams. In: ICDE, pp. 350–361 (2004)
16. Babu, S., Bizarro, P.: Adaptive Query Processing in the Looking Glass. In: CIDR, pp. 238–249 (2005)
17. Babu, S., Bizarro, P., DeWitt, D.J.: Proactive Re-optimization. In: SIGMOD Conference, pp. 107–118 (2005)
18. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive Ordering of Pipelined Stream Filters. In: SIGMOD Conference, pp. 407–418 (2004)
19. Babu, S., Munagala, K., Widom, J., Motwani, R.: Adaptive Caching for Continuous Queries. In: ICDE, pp. 118–129 (2005)
20. Babu, S., Srivastava, U., Widom, J.: Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM Trans. Database Syst.* 29(3), 545–580 (2004)
21. Babu, S., Widom, J.: StreaMon: An Adaptive Engine for Stream Query Processing. In: SIGMOD Conference, pp. 931–932 (2004)
22. Bar-Yossef, Z., Jayram, T.S., Kumar, R., Sivakumar, D., Trevisan, L.: Counting Distinct Elements in a Data Stream. In: Rolim, J.D.P., Vadhan, S.P. (eds.) RANDOM 2002. LNCS, vol. 2483, pp. 1–10. Springer, Heidelberg (2002)
23. Barbará, D., DuMouchel, W., Faloutsos, C., Haas, P.J., Hellerstein, J.M., Ioannidis, Y.E., Jagadish, H.V., Johnson, T., Ng, R.T., Poosala, V., Ross, K.A., Sevcik, K.C.: The New Jersey Data Reduction Report. *IEEE Data Eng. Bull.* 20(4), 3–45 (1997)
24. Belussi, A., Boucelma, O., Catania, B., Lassoued, Y., Podestà, P.: Towards Similarity-Based Topological Query Languages. In: Grust, T., Höpfner, H., Illarramendi, A., Jablonski, S., Fischer, F., Müller, S., Patranjan, P.-L., Sattler, K.-U., Spiliopoulou, M., Wijsen, J. (eds.) EDBT 2006. LNCS, vol. 4254, pp. 675–686. Springer, Heidelberg (2006)
25. Bizarro, P., Babu, S., DeWitt, D.J., Widom, J.: Content-Based Routing: Different Plans for Different Data. In: VLDB, pp. 757–768 (2005)
26. Börzsönyi, S., Kossmann, D., Stocker, K.: The Skyline Operator. In: ICDE, pp. 421–430 (2001)
27. Braumandl, R., Keidl, M., Kemper, A., Kossmann, D., Kreutz, A., Seltzsam, S., Stocker, K.: ObjectGlobe: Ubiquitous Query Processing on the Internet. *VLDB J.* 10(1), 48–71 (2001)

28. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring Streams - A New Class of Data Management Applications. In: VLDB, pp. 215–226 (2002)
29. Catania, B., Guerrini, G.: Towards Adaptively Approximated Search in Distributed Architectures. In: Vakali, A., Jain, L.C. (eds.) *New Directions in Web Data Management 1. Studies in Computational Intelligence*, vol. 331, pp. 171–212. Springer, Heidelberg (2011)
30. Chakrabarti, K., Garofalakis, M.N., Rastogi, R., Shim, K.: Approximate Query Processing using Wavelets. VLDB J. 10(2-3), 199–223 (2001)
31. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: CIDR (2003)
32. Chaudhuri, S., Das, G., Narasayya, V.R.: Optimized Stratified Sampling for Approximate Query Processing. ACM Trans. Database Syst. 32(2), 9 (2007)
33. Chaudhuri, S., Ganti, V., Kaushik, R.: A Primitive Operator for Similarity Joins in Data Cleaning. In: ICDE, p. 5 (2006)
34. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In: SIGMOD Conference, pp. 379–390 (2000)
35. Ciaccia, P., Patella, M.: PAC Nearest Neighbor Queries: Approximate and Controlled Search in High-Dimensional and Metric Spaces. In: ICDE, p. 244 (2000)
36. Considine, J., Hadjieleftheriou, M., Li, F., Byers, J.W., Kollios, G.: Robust Approximate Aggregation in Sensor Data Management Systems. ACM Trans. Database Syst. 34(1) (2009)
37. Considine, J., Li, F., Kollios, G., Byers, J.W.: Approximate Aggregation Techniques for Sensor Databases. In: ICDE, pp. 449–460 (2004)
38. Cormode, G., Garofalakis, M.N.: Sketching Streams Through the Net: Distributed Approximate Query Tracking. In: VLDB, pp. 13–24 (2005)
39. Corral, A., Cañadas, J., Vassilakopoulos, M.: Approximate Algorithms for Distance-Based Queries in High-Dimensional Data Spaces Using R-Trees. In: Manolopoulos, Y., Návrat, P. (eds.) ADBIS 2002. LNCS, vol. 2435, pp. 163–176. Springer, Heidelberg (2002)
40. Das, A., Gehrke, J., Riedewald, M.: Approximate Join Processing Over Data Streams. In: SIGMOD Conference, pp. 40–51 (2003)
41. Das, G., Gunopulos, D., Koudas, N., Sarkas, N.: Ad-hoc Top-k Query Answering for Data Streams. In: VLDB, pp. 183–194 (2007)
42. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining Stream Statistics over Sliding Windows. SIAM J. Comput. 31(6), 1794–1813 (2002)
43. Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J.M., Hong, W.: Model-Driven Data Acquisition in Sensor Networks. In: VLDB, pp. 588–599 (2004)
44. Deshpande, A., Ives, Z.G., Raman, V.: Adaptive Query Processing. *Foundations and Trends in Databases* 1(1), 1–140 (2007)
45. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate Record Detection: A Survey. IEEE Trans. Knowl. Data Eng. 19(1), 1–16 (2007)
46. Eurviriyankul, K., Paton, N.W., Fernandes, A.A.A., Lynden, S.J.: Adaptive Join Processing in Pipelined Plans. In: EDBT, pp. 183–194 (2010)
47. Gedik, B., Wu, K.L., Yu, P.S., Liu, L.: Adaptive Load Shedding for Windowed Stream Joins. In: CIKM, pp. 171–178 (2005)
48. Gedik, B., Wu, K.L., Yu, P.S., Liu, L.: CPU Load Shedding for Binary Stream Joins. Knowl. Inf. Syst. 13(3), 271–303 (2007)

49. Gedik, B., Wu, K.L., Yu, P.S., Liu, L.: GrubJoin: An Adaptive, Multi-Way, Windowed Stream Join with Time Correlation-Aware CPU Load Shedding. *IEEE Trans. Knowl. Data Eng.* 19(10), 1363–1380 (2007)
50. Gibbons, P.B., Matias, Y.: Synopsis Data Structures for Massive Data Sets. In: Abello, J.M., Vitter, J.S. (eds.) *External Memory Algorithms*, pp. 39–70. American Mathematical Society (1999)
51. Goldberg, D.E.: *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
52. Gounaris, A., Smith, J., Paton, N.W., Sakellariou, R., Fernandes, A.A.A., Watson, P.: Adaptive Workload Allocation in Query Processing in Autonomous Heterogeneous Environments. *Distributed and Parallel Databases* 25(3), 125–164 (2009)
53. Gravano, L., Ipeiritos, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate String Joins in a Database (Almost) for Free. In: *VLDB*, pp. 491–500 (2001)
54. Guha, S., Jagadish, H.V., Koudas, N., Srivastava, D., Yu, T.: Approximate XML Joins. In: *SIGMOD Conference*, pp. 287–298 (2002)
55. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: Ranked Keyword Search over XML Documents. In: *SIGMOD Conference*, pp. 16–27 (2003)
56. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: *SIGMOD Conference*, pp. 47–57 (1984)
57. Han, D., Wang, G., Xiao, C., Zhou, R.: Load Shedding for Window Joins over Streams. *J. Comput. Sci. Technol.* 22(2), 182–189 (2007)
58. Ilyas, I.F., Aref, W.G., Elmagarmid, A.K., Elmongui, H.G., Shah, R., Vitter, J.S.: Adaptive Rank-aware Query Optimization in Relational Databases. *ACM Trans. Database Syst.* 31(4), 1257–1304 (2006)
59. Ilyas, I.F., Beskales, G., Soliman, M.A.: A Survey of Top-*k* Query Processing Techniques in Relational Database Systems. *ACM Comput. Surv.* 40(4) (2008)
60. Ioannidis, Y.E., Kang, Y.: Randomized Algorithms for Optimizing Large Join Queries. *SIGMOD Rec.* 19, 312–321 (1990)
61. Ioannidis, Y.E., Ng, R.T., Shim, K., Sellis, T.K.: Parametric Query Optimization. In: *VLDB*, pp. 103–114 (1992)
62. Ioannidis, Y.E., Poosala, V.: Histogram-Based Approximation of Set-Valued Query-Answers. In: *VLDB*, pp. 174–185 (1999)
63. Ives, Z.G., Deshpande, A., Raman, V.: Adaptive Query Processing: Why, How, When, and What Next? In: *VLDB*, pp. 1426–1427 (2007)
64. Ives, Z.G., Florescu, D., Friedman, M., Levy, A.Y., Weld, D.S.: An Adaptive Query Execution System for Data Integration. In: *SIGMOD Conference*, pp. 299–310 (1999)
65. Ives, Z.G., Halevy, A.Y., Weld, D.S.: Adapting to Source Properties in Processing Data Integration Queries. In: *SIGMOD Conference*, pp. 395–406 (2004)
66. Jiao, Y.: Maintaining Stream Statistics over Multiscale Sliding Windows. *ACM Trans. Database Syst.* 31, 1305–1334 (2006)
67. Kabra, N., DeWitt, D.J.: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In: *SIGMOD Conference*, pp. 106–117 (1998)
68. Kadlag, A., Wanjari, A.V., Freire, J.-L., Haritsa, J.R.: Supporting Exploratory Queries in Databases. In: Lee, Y., Li, J., Whang, K.-Y., Lee, D. (eds.) *DASFAA 2004*. LNCS, vol. 2973, pp. 594–605. Springer, Heidelberg (2004)
69. Kang, J., Naughton, J.F., Viglas, S.: Evaluating Window Joins over Unbounded Streams. In: *ICDE*, pp. 341–352 (2003)
70. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by Simulated Annealing. *Science* 220, 671–680 (1983)

71. Kossmann, D., Ramsak, F., Rost, S.: Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In: VLDB, pp. 275–286 (2002)
72. Koudas, N., Li, C., Tung, A.K.H., Vernica, R.: Relaxing Join and Selection Queries. In: VLDB, pp. 199–210 (2006)
73. Koudas, N., Sarawagi, S., Srivastava, D.: Record Linkage: Similarity Measures and Algorithms. In: SIGMOD Conference, pp. 802–803 (2006)
74. Koudas, N., Srivastava, D.: Approximate Joins: Concepts and Techniques. In: VLDB, p. 1363 (2005)
75. Kulkarni, D., Ravishankar, C.V.: iJoin: Importance-Aware Join Approximation Over Data Streams. In: Ludäscher, B., Mamoulis, N. (eds.) SSDBM 2008. LNCS, vol. 5069, pp. 541–548. Springer, Heidelberg (2008)
76. Lee, D.: Query Relaxation for XML Model. Ph.D. thesis. University of California (2002)
77. Lengu, R., Missier, P., Fernandes, A.A.A., Guerrini, G., Mesiti, M.: Time-completeness Trade-offs in Record Linkage using Adaptive Query Processing. In: EDBT, pp. 851–861 (2009)
78. Li, Q., Shao, M., Markl, V., Beyer, K.S., Colby, L.S., Lohman, G.M.: Adaptively Reordering Joins during Query Execution. In: ICDE, pp. 26–35 (2007)
79. Liu, B., Zhu, Y., Jbantova, M., Momberger, B., Rundensteiner, E.A.: A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In: VLDB, pp. 1338–1341 (2005)
80. Liu, H., Wang, X., Yang, Y.: Comments on “An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams”. *ACM Trans. Database Syst.* 35(2) (2010)
81. Liu, X., Dong, X.L., Ooi, B.C., Srivastava, D.: Online Data Fusion. In: VLDB (2011)
82. Liu, Y., Li, J., Gao, H., Fang, X.: Enabling epsilon-Approximate Querying in Sensor Networks. *PVLDB* 2(1), 169–180 (2009)
83. Lu, H., Zhou, Y., Haustad, J.: Continuous Skyline Monitoring Over Distributed Data Streams. In: Gertz, M., Ludäscher, B. (eds.) SSDBM 2010. LNCS, vol. 6187, pp. 565–583. Springer, Heidelberg (2010)
84. Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously Adaptive Continuous Queries over Streams. In: SIGMOD Conference, pp. 49–60 (2002)
85. Manku, G.S., Motwani, R.: Approximate Frequency Counts over Data Streams. In: VLDB, pp. 346–357 (2002)
86. Marian, A., Amer-Yahia, S., Koudas, N., Srivastava, D.: Adaptive Processing of Top-k Queries in XML. In: ICDE, pp. 162–173 (2005)
87. Markl, V., Raman, V., Simmen, D.E., Lohman, G.M., Pirahesh, H.: Robust Query Processing through Progressive Optimization. In: SIGMOD Conference, pp. 659–670 (2004)
88. Metwally, A., Agrawal, D., Abbadi, A.E.: An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams. *ACM Trans. Database Syst.* 31(3), 1095–1133 (2006)
89. Mishra, C., Koudas, N.: Interactive Query Refinement. In: EDBT, pp. 862–873 (2009)
90. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G.S., Olston, C., Rosenstein, J., Varma, R.: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In: CIDR (2003)
91. Mouratidis, K., Bakiras, S., Papadias, D.: Continuous Monitoring of Top-k Queries over Sliding Windows. In: SIGMOD Conference, pp. 635–646 (2006)
92. Mozafari, B., Zaniolo, C.: Optimal Load Shedding with Aggregates and Mining Queries. In: ICDE, pp. 76–88 (2010)

93. Munagala, K., Srivastava, U., Widom, J.: Optimization of Continuous Queries with Shared Expensive Filters. In: PODS, pp. 215–224 (2007)
94. Olston, C., Jiang, J., Widom, J.: Adaptive Filters for Continuous Queries over Distributed Data Streams. In: SIGMOD Conference, pp. 563–574 (2003)
95. Pan, L., Luo, J., Li, J.: Probing Queries in Wireless Sensor Networks. In: ICDCS, pp. 546–553 (2008)
96. Papadias, D., Arkoumanis, D.: Approximate Processing of Multiway Spatial Joins in Very Large Databases. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 179–196. Springer, Heidelberg (2002)
97. Papadias, D., Mantzourogiannis, M., Kalnis, P., Mamoulis, N., Ahmad, I.: Content-based Retrieval using Heuristic Search. In: SIGIR, pp. 168–175. ACM, New York (1999)
98. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive Skyline Computation in Database Systems. *ACM Trans. Database Syst.* 30(1), 41–82 (2005)
99. Poosala, V., Ganti, V., Ioannidis, Y.E.: Approximate Query Answering using Histograms. *IEEE Data Eng. Bull.* 22(4), 5–14 (1999)
100. Raman, V., Deshpande, A., Hellerstein, J.M.: Using State Modules for Adaptive Query Processing. In: ICDE, p. 353 (2003)
101. Reiss, F., Hellerstein, J.M.: Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In: VLDB (2004)
102. Rundensteiner, E.A., Ding, L., Sutherland, T.M., Zhu, Y., Pielech, B., Mehta, N.: CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. In: VLDB, pp. 1353–1356 (2004)
103. Rusu, F., Dobra, A.: Sketching Sampled Data Streams. In: ICDE, pp. 381–392 (2009)
104. Sanz, I., Mesiti, M., Guerrini, G., Llavori, R.B.: Fragment-based Approximate Retrieval in Highly Heterogeneous XML Collections. *Data Knowl. Eng.* 64(1), 266–293 (2008)
105. Sarawagi, S., Kirpal, A.: Efficient Set Joins on Similarity Predicates. In: SIGMOD Conference, pp. 743–754 (2004)
106. Sarkas, N., Das, G., Koudas, N., Tung, A.K.H.: Categorical Skylines for Streaming Data. In: SIGMOD Conference, pp. 239–250 (2008)
107. Shah, M.A., Hellerstein, J.M., Chandrasekaran, S., Franklin, M.J.: Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In: ICDE, pp. 25–36 (2003)
108. Silberstein, A., Braynard, R., Ellis, C.S., Munagala, K., Yang, J.: A Sampling-Based Approach to Optimizing Top-k Queries in Sensor Networks. In: ICDE, p. 68 (2006)
109. Silva, Y.N., Aref, W.G., Ali, M.H.: The similarity join database operator. In: ICDE, pp. 892–903 (2010)
110. Skordylis, A., Trigoni, N., Guitton, A.: A Study of Approximate Data Management Techniques for Sensor Networks. In: *Intelligent Solutions in Embedded Systems*, pp. 1–12 (2006)
111. Spiegel, J., Polyzotis, N.: TuG Synopses for Approximate Query Answering. *ACM Trans. Database Syst.* 34(1) (2009)
112. Srivastava, U., Widom, J.: Memory-Limited Execution of Windowed Stream Joins. In: VLDB, pp. 324–335 (2004)
113. Sun, S., Huang, Z., Zhong, H., Dai, D., Liu, H., Li, J.: Efficient Monitoring of Skyline Queries over Distributed Data Streams. *Knowl. Inf. Syst.* 25(3), 575–606 (2010)
114. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient Progressive Skyline Computation. In: VLDB, pp. 301–310 (2001)
115. Tao, Y., Papadias, D.: Maintaining Sliding Window Skylines on Data Streams. *IEEE Trans. Knowl. Data Eng.* 18(2), 377–391 (2006)

116. Tatbul, N., Çetintemel, U., Zdonik, S.B., Cherniack, M., Stonebraker, M.: Load Shedding in a Data Stream Manager. In: VLDB, pp. 309–320 (2003)
117. Tatbul, N., Zdonik, S.B.: Window-Aware Load Shedding for Aggregation Queries over Data Streams. In: VLDB, pp. 799–810 (2006)
118. Theobald, M., Bast, H., Majumdar, D., Schenkel, R., Weikum, G.: TopX: Efficient and Versatile Top-k Query Processing for Semistructured Data. VLDB J. 17(1), 81–115 (2008)
119. Tian, F., DeWitt, D.J.: Tuple Routing Strategies for Distributed Eddies. In: VLDB, pp. 333–344 (2003)
120. Tirthapura, S., Xu, B., Busch, C.: Sketching Asynchronous Data streams over Sliding Windows. Distributed Computing 20(5), 359–374 (2008)
121. Urhan, T., Franklin, M.J., Amsaleg, L.: Cost Based Query Scrambling for Initial Delays. In: SIGMOD Conference, pp. 130–141 (1998)
122. Vitter, J.S., Wang, M.: Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In: SIGMOD Conference, pp. 193–204 (1999)
123. Wang, J., Li, G., Feng, J.: Trie-Join: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints. PVLDB 3(1), 1219–1230 (2010)
124. Weis, M., Naumann, F.: DogmatiX Tracks down Duplicates in XML. In: SIGMOD Conference, pp. 431–442 (2005)
125. Wilschut, A.N., Apers, P.M.G.: Dataflow Query Execution in a Parallel Main-Memory Environment. In: PDIS, pp. 68–77 (1991)
126. Wu, J., Tan, K.-L., Zhou, Y.: QoS-Oriented Multi-Query Scheduling Over Data Streams. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 215–229. Springer, Heidelberg (2009)
127. Yang, Y., Krämer, J., Papadias, D., Seeger, B.: HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. IEEE Trans. Knowl. Data Eng. 19(3), 398–411 (2007)
128. Yi, K., Li, F., Cormode, G., Hadjieleftheriou, M., Kollios, G., Srivastava, D.: Small Synopses for Group-by Query Verification on Outsourced Data Streams. ACM Trans. Database Syst. 34(3) (2009)
129. Yu, H., Hwang, S.-w., Chang, K.C.C.: Enabling Ad-hoc Ranking for Data Retrieval. In: ICDE, pp. 514–515 (2005)
130. Zhang, Z., Hwang, S.-w., Chang, K.C.C., Wang, M., Lang, C.A.3., Chang, Y.C.: Boolean Ranking: Querying a Database by k-constrained Optimization. In: SIGMOD Conference, pp. 359–370 (2006)
131. Zhou, X., Gaugaz, J., Balke, W.T., Nejdl, W.: Query Relaxation using Malleable Schemas. In: SIGMOD Conference, pp. 545–556 (2007)
132. Zimbrão, G., de Souza, J.M.: A Raster Approximation For Processing of Spatial Joins. In: VLDB, pp. 558–569 (1998)

Chapter 11

Querying Conflicting Web Data Sources

Gilles Nachouki, Mohamed Quafafou, Omar Boucelma,
and François-Marie Colonna

Abstract. Over the last twenty years, information integration has received considerable efforts from both industry and academia. Approaches to information integration developed so far can be categorized as follows: (1) first-generation approaches, that require the definition of a global schema and a semantic integration which should be performed upfront (before query execution); (2) second-generation approaches, well illustrated by the *dataspace management* concept, which promote a *pay-as-you-go* data integration. The first category has led to well known mediation approaches such as GAV (Global as View), LAV (Local as View), GLAV (Generalized Local As View), BAV (Both As View), and BGLAV (BYU Global-Local-as-View). Approaches pertaining to the second category are geared towards the development of dataspace management systems and are currently gaining a lot of attention. In this chapter we are interested in exploiting both types of approaches in querying conflicting data spread over multiple web sources. To this aim, first we show how an XML-based BGLAV approach can handle these conflicting data sources, then we describe how the same problem can be addressed by using the Multi Fusion Approach (MFA), an approach pertaining to second-generation techniques. Both BGLAV and MFA are illustrated in using genomic data sources accessible through the Web.

Gilles Nachouki
LINA-UMR CNRS 6241, Nantes University, France
e-mail: Gilles.Nachouki@univ-nantes.fr

Mohamed Quafafou · Omar Boucelma
LSIS-UMR CNRS 6168, Aix-Marseille University, France
e-mail: {mohamed.quafafou,omar.boucelma}@lsis.org

François-Marie Colonna
Institut Supérieur de l'Electronique et du Numérique, France
e-mail: francois-marie.colonna@isen.fr

11.1 Introduction

Over the past two decades, the database communities (both industry and academia), have intensively addressed data integration problems. A first-generation category of approaches and systems has been developed, and significant contributions have been made in various subtopics [16] such as: source descriptions, schema mappings, query reformulation, and incomplete information modeling. Approaches pertaining to this first category require upfront semantic integration, that is a global/mediation schema needs to be supplied beforehand. As an example, in mediation approaches such as GAV (Global as View) [12], LAV(Local as View) [23], GLAV(Generalized Local As View) [11] or BAV(Both As View) [5], the *mediator* provides the user with a global schema and allows him/her to access heterogeneous data sources providing the illusion to access a single local database. The *wrapper*, another component of mediation systems, plays the role of an interface between the mediator and the data sources: it receives queries from a mediator and uses its own knowledge (source descriptions and mapping rules) in order to access the data sources.

As an example, in LAV the content of each data source is expressed in terms of a view over the global schema. Mapping rules associate a query over the global schema to each element of the local sources. On the other hand, in GAV each element of the global schema is expressed in terms of a view over the data sources. Each mapping rule associates a query over a local source to each element in the global schema.

The main component of first-generation data integration systems is the query rewriting module; it explores a set of mappings in order to rewrite queries, expressed upon the global schema, in terms of local sources' schemas. The complexity of the query rewriting phase depends on how the global schema is defined. For example, GAV query rewriting is very simple since the elements in the global schema are defined in terms of the source schemas [22]. In this case, query rewriting simply consists in unfolding the definitions of the elements in the global schema. However, in this case, adding a new source to the data integration system is not trivial. The new source may indeed have an impact on the definition of various elements of the global schema, whose associated views need to be redefined. On the other hand, LAV query rewriting is not straightforward (exponential time complexity) and many rewriting algorithms have been developed; among them, we recall the Bucket, Inverse Rules and MiniCon algorithms [14], to cite a few. At the same time, however, the LAV approach favors the extensibility of the system: adding a new source simply means enriching the mapping with a new assertion, without other changes.

In general, query rewriting works well assuming that the schema of the local source is known a-priori and static. Unfortunately, such assumptions are not satisfied by data managed in several new processing environments, where data sources to be integrated are selected and combined on-demand. New second-generation data integration approaches have therefore been proposed. Dataspaces and Dataspace Management Systems (DSMSs), described in Chapter 12, represent a significant

example of second-generation integration approaches. A dataspace has the following features [10]:

1. It can handle data with different formats accessible through different interfaces (e.g., database system, data file etc.). A dataspace is designed to support all kinds of data (e.g., structured, semi-structured or unstructured data etc.).
2. It provides an interface to search, retrieve, update and manage a dataspace, through a DSMS. Unlike a Database Management System (DBMS), a DSMS does not completely control its data; but it offers various levels of services in order to return the best answer.
3. A dataspace provides all software in order to improve data integration.

The requirements and the architecture of a DSMS is presented in detail in [15]. DSMSs promote a *pay-as-you-go* integration system where “the system starts with very few (or inaccurate) semantic mappings and these mappings are improved over time as deemed necessary” [33, 19].

In this chapter, we exploit both first-generation and second-generation approaches in describing how to query multiple heterogeneous conflicting web data sources. To this aim, we assume that web data sources are represented in XML; a concrete example drawn from Genomics will be used to illustrate the proposed concepts. Two different data integration techniques will be considered. The first relies on a mediation approach [8, 7] based on BGLAV [36],¹ first defined for relational data and then extended to deal with XML data sources [7]. BGLAV has been proposed to overcome both GAV and LAV limitations. BGLAV improves both GAV and LAV because the global schema remains unchanged when a data source is added or updated. BGLAV, as a first-generation data integration approach, is characterized by the presence of a global schema and the need to specify a set of predefined, hard-coded correspondence queries (mappings), which specify how to solve conflicts among local sources. Mappings have to be specified before submitting queries to the mediator, which is in charge of the translation process, that leads to the generation of the sub-queries posed over local schemas.

In the web context, the definition of such a global schema and the maintenance of mappings become cumbersome. As an alternative approach to BGLAV we therefore consider the Multi-Fusion Approach (MFA), a data fusion method developed by Nachouki et al [26]. MFA, although not being directly inspired by the dataspace management systems concepts, relates to the second-generation of data integration approaches. MFA does not assume the definition of a global schema beforehand; rather, it relies on the Multi-data source Fusion Language (MFL) for the definition of a multi-data source schema (a kind of dataspace) and the retrieval of data issued from conflicting sources. With MFA, conflicts (i.e., assertions or mappings) between data sources can be specified by a (skilled) user who has some domain knowledge. Users have the possibility to refine these conflicts later in order to increase gradually the accuracy of their queries. In MFA, unlike BGLAV, retrieving data from conflicting data sources is established using semantic queries, which may include conflicting elements in their bodies. MFA query processing consists in resolving

¹ BYU Global-Local-as-View, where BYU stands for Brigham Young University.

conflicts and then in decomposing the query into a set of sub-queries to be sent to data sources for execution.

The contributions of this chapter are therefore twofold: (1) it fully illustrates the various phases of the data integration process, namely, schema integration, data reconciliation (fusion), query rewriting, etc., in taking into account both a traditional and a more advanced integration approach; (2) it demonstrates the application of the presented concepts and algorithms in using a real data set, drawn from the Genomics domain. The chapter is organized as follows. Section 11.2 describes conflicts between (biological) data sources and provides a taxonomy of conflicts in a general context. Section 11.3 illustrates how to process mediated queries in BGLAV. Section 11.4 is devoted to MFA. An overall example taken from the biological domain is provided in Section 11.5. Finally, Section 11.6 presents some discussion and conclusions and outlines future work.

11.2 Conflicting Web Data Sources

In the following, we first briefly classify conflicts that may arise from multiple data sources to be integrated. Then, we discuss various types of conflicts arising from the life science domain through an example and we present some assumptions about conflict representation upon which the results presented in the chapter rely.

11.2.1 Overview of Conflict Types

In most data integration examples covered in the literature, data sources present various types of heterogeneity, concerning differences in names, data structures, types, scale, just to cite a few. This is due to the fact that several perceptions of the same real world lead to different data modeling of the same entity. In order to integrate a set of data sources, all such conflicts have first to be solved. Conflicts can be classified as follows [35]:

- *Data conflicts*, referring to differences among definitions, such as attribute types, formats, or precision.
- *Structural conflicts*, arising from the description of the same concept in different ways and in different data sources. For example, a concept can be defined as an attribute in a relational schema $Sch1$ and as a relation in another schema $Sch2$.
- *Descriptive conflicts*, including the usage of different names for the same entity (e.g., homonymous, synonymous), identity conflicts (e.g., a person is identified by a number in $Sch1$ and social-security number in $Sch2$), scale conflicts (e.g., salaries are given in Dollar and Euro respectively in $Sch1$ and $Sch2$).
- *Abstraction conflicts*, concerning the presence of generalization/specialization concepts (e.g., the concept of employee in $Sch1$ generalizes the concept of

teacher in *Sch2*) and aggregation (e.g., date of birth in *Sch1* is a string while it is composed of three fields month, day and year in *Sch2*).

- *Semantic conflicts*, referring to differences and similarities in the meaning of concepts in the data sources.

Many works have investigated semantic conflicts in the literature. In [6], authors propose an algorithm which takes two schemas as input and returns the mappings that identify corresponding concepts in the two schemas, namely the concepts with the same or the closest meaning. In [32], authors provide a survey of different approaches to automatic schema matching.

11.2.2 *Conflicting Data in Life Sciences*

Many data management applications require the integration of data from multiple sources [9], often available on the Web as XML documents. For instance, in the field of biology, the number of data sources and tools available in the Web has grown in recent years. This huge augmentation of data sources has led to a deep heterogeneity between data sources and to a variety of interfaces. Until today, the reconciliation between data sources is performed manually by biologists. Scientific investigations on *Genes* or *Proteins* -for annotations or predictions- or information retrieval from scientific publications (journals, conferences, etc.) often lead researchers to submit queries to several (yet heterogeneous) data sources that are available on the Web. As an example, Mootha et al. [25] discovered one of the genes responsible of *Leigh* syndrome by integrating both expression, genomic and sub-cellular localization data.

In the biological domain, the same/identical information may be stored using distinct formats or structures such as ASN 1.0 [2] or Fasta [30], HTML or XML, leading to some data conflicts. As an example, Figure 11.1 shows a description of *ILB12*, a gene that encodes a subunit of *interleukin 12*, which is one of the regulatory proteins that are released by cells of the immune system. As illustrated, the same entity, *ILB12*, is described by means of several heterogeneous schemas.

Semantic conflicts are also quite common in the life sciences domain. For example, in [34] two definitions of the concept of *gene* have been compared: in GDB [18], a gene is a DNA fragment that can be transcribed and translated into a protein; for Genbank [3] and GSDB [21], a gene is a “*DNA region of biological interest with a name and that carries a genetic trait or phenotype*”, which includes nonstructural coding DNA regions like intron, promoter and enhancer. There is a clear semantic distinction between those two notions of gene but both are still being used, hence adding another level of complexity into the data integration process. Another term that comes with multiple meanings is *protein function*, that could be defined either as a biochemical function (e.g., enzyme catalysis), a genetic function (e.g., transcription repressor), a cellular function (e.g. scaffold), or as a physiological function (e.g., signal transducer).

```

<!-- HTML -->
<!------->
<table border="0" width="100" cellpadding="1" cellspacing="1">
  <tr>
    <td nowrap="nowrap">Entry name</td>
    <td width="100">
      <b>IL12B\_HUMAN</b>
    </td>
  </tr>
  <tr>
    <td nowrap="nowrap">Primary accession number</td>
    <td>
      <b>P29460</b>
    </td>
  </tr>
  <tr>
    <td nowrap="nowrap">Integrated into Swiss-Prot on</td>
    <td>April 1, 1993</td>
  </tr>
</table>

```

```

<!-- ASN 1.0 -->
<!------->
Seq-entry ::= set {
  descr {title "Interleukin-12 subunit beta" ,
    update-date std {year 1991 ,month 5 ,day 17} ,
    source {org {taxname "Homo sapiens" , common "human" ,
      db {db "taxon" , tag id 9606}
    }
  }
}

```

```

<!-- FASTA -->
<!------->
>IL12B|chr5|-|158674369|158690059
GATTACAAAGAAGAGTTTATTAGTTCAGCCTCAGAATGCAAAAATAAA
%TAATAAATAAACAAACAGGAAACAAATGTAATCACTTTACAGAGCGCAC
ATACATTACTTAAAAGTAGCACCTTCATGGAGCCATATTTCTGGTCATA
.....

```

```

<!-- XML -->
<!------->
<SNPPER-RPC SOURCE="*RPCSERV-NAME*" VERSION="$Revision: 1.38$"
  GENOME="hg17" DBSNP="123">
  <GENEINFO>
    <GENEID>16348</GENEID>
    <NAME>IL12B</NAME>
    <CHROM>chr5</CHROM>
    <STRAND>-</STRAND>
    <TRANSCRIPT>
      <START>158674369</START>
      <END>158690059</END>
    </TRANSCRIPT>
  </GENEINFO>
</SNPPER-RPC>

```

Fig. 11.1 Structural conflicts between Genomics data sources.

11.2.3 Assumptions about Conflict Representation

In this chapter, we will consider the most important types of conflicts detailed above, i.e., data conflicts, structural conflicts, descriptive conflicts, and semantic conflicts. In order to deal with conflicts between data sources, we assume that all data sources schemas are represented according to a common data model (e.g., XML schema, DTD, relational model, etc.). In this chapter, we consider XML documents; we assume that schema information is represented as a DTD document. To simplify the discussion, we rely on a tree-based representation of both XML documents and schema information. Under this assumption, semantic conflicts between elements are specified using the concept of contexts of elements. The *context* of an element E is the set of elements connected to E by a parent-child or ancestor-descendant relationship. The context of an element is therefore the set of elements semantically depending on it. In other words, if a node $E2$ is a child of a node $E1$, the element $E2$ has to be interpreted in the scope of $E1$'s meaning. As a consequence, different occurrences of the same label do not have the same meaning: for example, label *Name* may appear several times in the same tree under different contexts, thus representing different semantic entities.

11.3 Mediating Biological Conflicting Data with BGLAV

In this section, we illustrate the BGLAV approach for querying conflicting web data sources.

11.3.1 BGLAV Overview

BGLAV was proposed initially by Li Xu et al. in [36] in the context of relational databases. We adapted this approach for mediating web data sources, represented as XML documents and optionally conflicting, queried through XQuery [7]. BGLAV [36] alleviates GAV and LAV drawbacks in defining source-to-target mappings based on a predefined conceptual target schema (global schema), which is specified independently of any of the sources. More precisely, in a GAV approach, changes in information sources or adding a new information source require revisions of the global schema and mappings between the global schema and source schemas. In a LAV approach, automating query reformulation is hard (i.e., it has exponential time complexity with respect to query and source schema definitions).

To resolve these problems, BGLAV offers an alternative point of view in defining source-to-target mappings based on a predefined conceptual global schema. In particular, the global schema in BGLAV is ontologically specified, independently of any of the sources. BGLAV keeps the advantages of the two approaches GAV and LAV: GAV's simple query reformulation and LAV's scalability. Additionally, it is characterized by the following features:

- Each concept in a target schema (global schema) is predefined and independent of any source schema. In contrast, under GAV, Data Base Administrators (DBAs) revise the global schema to include all concepts represented inside any source; under LAV, DBAs adjust the source schemas such that they contain only source relations that can be described by views over the global schema.
- A set of source-to-target mappings maps a source schema to a target schema. Source and target schemas can use different structures and vocabularies.
- When a new local source becomes available (i.e., a change occurs), a source-to-target mapping must be created (or adjusted).

11.3.2 Query Processing in BGLAV

In this section, we highlight BGLAV query processing. First we provide some necessary background, then we illustrate the query rewriting steps. We start by introducing the concept of *correspondence query* between a source schema and a global schema. The idea is that of defining a correspondence, i.e., a mapping, between (a part of) the source schema and (a part of) the global schema, taking into account existing conflicts.

Definition 1 (Correspondence query). Let S_l be a source schema and G be a global schema. Let T_{S_l} and T_G be two sub-trees belonging to S_l and G respectively. A correspondence query (or mapping query) is defined as a set of transformation operations which are applied to T_{S_l} and produce a new tree denoted by $T_{S_l'}$ whose elements are in direct correspondence with those of T_G . We assume that transformation operators are specified using XQuery.

Example 1. Consider the data sources S_1 and S_2 and the global schema G , represented in Figure 11.2. Let T_{S_1} and T_G be the sub-trees showed in the figure. What follows is an example of a correspondence query between S_1 and G :

```

<length>
  for $x in document(S1)/strands/dna/strand/length
  return $x/3
</length>
```

This query illustrates a scale conflict between the two elements *length* in G and S_1 . The following is an example of a correspondence query between S_2 and G :

```

<date_seq>
  for $x in document(S2)/genes_list/gene/strand/date_seq
  return concat($x/day, '/', $x/month, '/', $x/year)
</date_seq>
```

This second query illustrates a structural conflict due to the different representations of element *date* in G and S_2 , respectively.

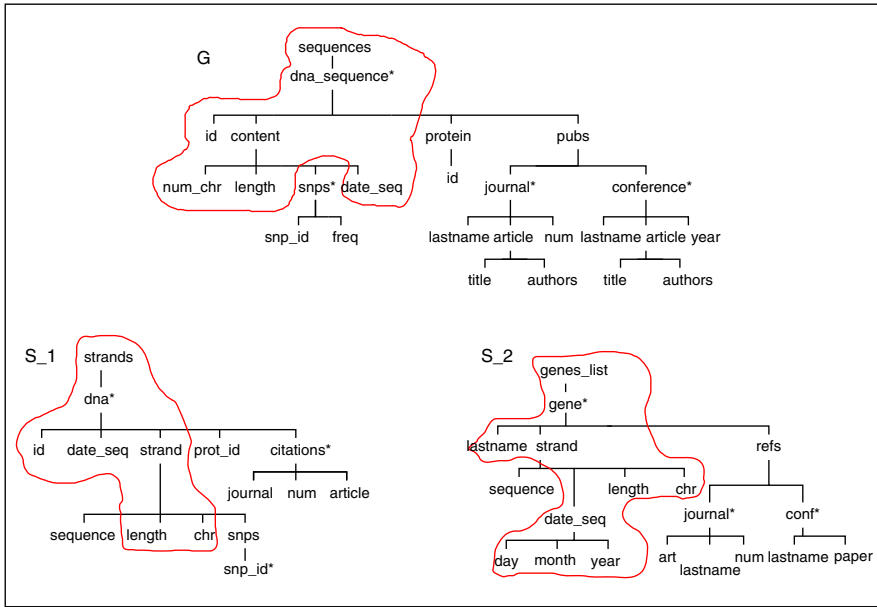


Fig. 11.2 Examples of queries in BGLAV.

Given a set of correspondence queries, from each local source it is possible to generate a derived schema, which is not (or at least partially) conflicting with G .

Definition 2 (Derived schema). Let S_i be a source schema and M_i be a set of correspondence queries that associate S_i to the global schema G . The schema V_i obtained by applying the mapping queries to S_i is called *derived schema*. The transformed elements in V_i are in direct correspondence with those of G .

Derived schemas satisfy the following property (inclusion dependency) with respect to the global schema.

Proposition 1 (Inclusion dependency). Let $V = \{V_i | i \in [1, n]\}$ be a set of derived schemas and G be the global schema. For each sub-tree T_G of G , there exists a subset of derived schemas $\bar{V} = \{V_{i_1}, \dots, V_{i_k}\}$ such that T_G corresponds to a set of sub-trees $\{T_{j,h}\}$ of $V_j \in \bar{V}$, $j \in \{i_1, \dots, i_k\}$. The instances of T_G and $T_{j,k}$ satisfy an inclusion dependency defined as follows: $I(T_{j,k}) \subseteq I(T_G)$, where $I(T)$ denotes the set of instances (i.e., XML documents) of the sub-tree T .

User queries can now be defined as follows.

Definition 3 (User query). Let G be the global schema, represented by a tree T_G . A query Q over G is expressed in XQuery over T_G and can be defined by a logical rule as $Q(T_G) :- T_1, T_2, \dots, T_p, C_Q$, where:

- T_i , $i \in [1, p]$, is a sub-tree of T_G , specified through an XQuery expression, in accordance with the following sentence: $\exists i \in [1, p]$, $\exists j \in [1, p]$, $i \neq j$, and T_i is a sub-tree of T_j ;
- C_Q is a set of conditions upon trees T_1, T_2, \dots, T_p , specified according to XQuery.

Depending on the relationship existing between the query tree and derived schemas, three distinct types of queries can be devised, as defined below. Different query types will lead to different choices during the query rewriting step.

Definition 4 (Completely and partially covered query). Let G be the global schema, represented by a tree T_G . Let $V = \{V_i | i \in [1, n]\}$ be a set of derived schema. Let $Q(T_G)$:- $T_1, T_2, \dots, T_p, C_Q$ be a query over T_G . Sub-trees T_i , $i \in [1; p]$, appearing in Q can be classified according to the three cases described below:

1. *Complete coverage*: T_i is completely covered by a sub-tree in all derived schemas in V .
2. *Partial coverage on some derived schemas*: T_i is partially covered by some derived schemas in V and completely covered by some others derived schemas in V ;
3. *Partial coverage on all derived schemas*: T_i is partially covered by all derived schemas in V .

In case of complete coverage, the answer is the union of the results coming from local sources. In case of partial coverage on some derived schemas, from some sources only a partial result can be retrieved. In case of partial coverage on all derived schemas, partial results should be joined (through XQuery) by means of the key elements they share in order to get query answers.

Figure 11.3 illustrates a case of partial coverage, which is quite common in the web context. The partial answers returned by the two data sources are joined by means of the common values of key elements (e.g., *id_gene* or *gene_id*).

Query processing is performed in mainly two steps [8, 7]:

1. The first step of the algorithm consists in identifying the correspondence queries that should be taken into account in processing the user query Q specified over the global schema. These correspondence queries will be used to define the queries to be executed against the local sources. The overall idea is that of trying to completely cover the query trees by joining local results obtained from correspondence queries together. Correspondence queries generating deriving schemas which are either totally or partially covering trees T_i in Q are taken into account. When the query is decomposed into sub-queries, each tree T_i in Q is then replaced by such correspondence queries, according to the local sources that should be accessed.
2. The second step consists in generating the query plan, which contains the set of sub-queries that access data sources in order to extract the results. Then, the results obtained from each data source are merged together (i.e., they are fused) and returned to the user. For detailed description of the algorithm that performs this step, we refer the reader to [7].

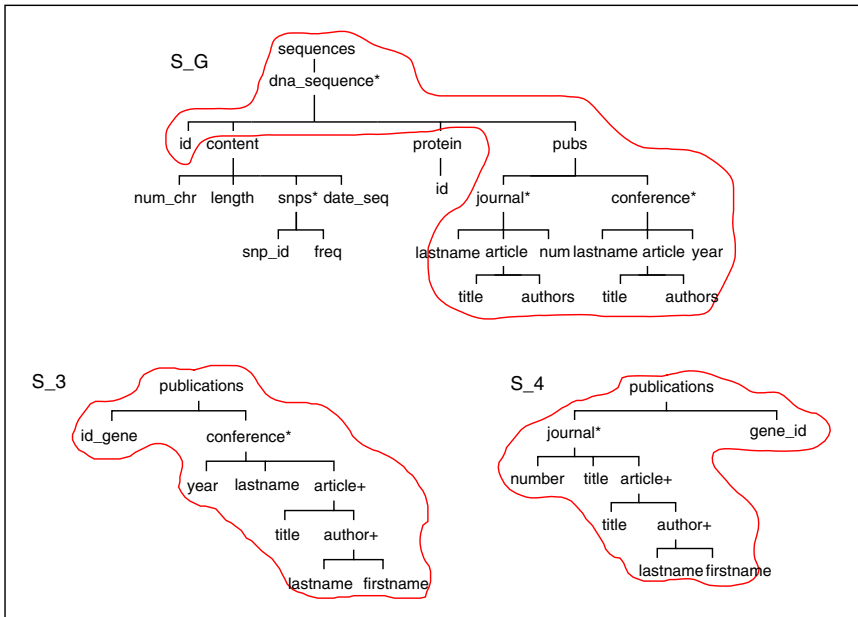


Fig. 11.3 Queries over partially covered global schema.

11.4 MFA - Multi-source Fusion Approach

In this section, we describe a second-generation of data integration approach and we show its use in querying conflicting web data sources.

11.4.1 MFA Overview

A lot of data sources are freely accessible on the Web and users often need to integrate them quickly without any help. Classical approaches based on mediator's reasoning do not facilitate the user's task since it is hard to unify data sources in a dynamic way. Rather, they assume a global mediated schema to model data sources. As such, these approaches often require an administrator to control the mediated schema. MFA differs from traditional approaches, like LAV, GAV, GLAV and BGLAV, in the two following aspects: (i) it does *not require a global schema* or ontology; (ii) mappings are established *only between data sources* (unlike other approaches). The motivation behind this approach is that, in some context, making calls to an administrator to check the mediated schema is not always required and it may be too onerous and restrictive. For example, in the web context, a user which wishes to integrate biological data is not forced to call an administrator to control

the mediated schema. MFA thus provides an inexpensive solution to a hard problem of data integration. By cons, this approach requires a minimal user knowledge about these sources and of their conflicts.

Under MFA, the user just locates its data sources (e.g., web sites), builds the multi-data source schema, and submits the queries. Submitting queries may be accomplished before resolving all the conflicts between data sources: users have the possibility to add (or refine) conflicts later in order to increase gradually the accuracy of their queries. This approach facilitates the integration of new data sources or deletion of an existing source. It also provides some languages that permit to define and retrieve data from multiple data sources while taking into account conflicts between sources.

A data integration system that follows the MFA approach is defined as a triple $\langle MS, S_i, M \rangle$, where:

- MS is the multi-data source schema;
- S_i is the set of data sources' schemas;
- M is the set of source-to-source mappings expressed as functionals $f: T_{S_1} \rightarrow T_{S_2}$; f maps elements s_1 appearing in the tree T_{S_1} corresponding to the source schema S_1 into elements s'_1 appearing in the tree T_{S_2} corresponding to the source schema S_2 .

MFA is based on the Multi-data source Fusion Language (MFL). MFL allows the definition and the retrieval of data originating from conflicting data sources, through the concept of *multi-data source* as a set of local sources. MFL is a simple and powerful language. It facilitates queries over conflicting data sources and controls the semantics expressed in user queries. For each query posed over a multi-data source schema, MFL will search for conflicts in the query body. If no conflict is detected, the query is validated and executed; otherwise three cases may arise (for more details, we refer the reader to [26, 27, 28]):

1. conflicts can be solved at query execution time by using the available source-to-source mappings: in this case, the query is validated and executed;
2. conflicts cannot be solved (e.g., in the case of homonymies): the query is rejected;
3. only a subset of the conflicts can be solved: in this case, only a part of the query, related to the solved conflicts, will be executed and the results will be returned to the user with a warning message informing him/her of the detected conflict nature.

MFL provides two sub-languages [26]: MDL, the *Multi-data source Definition Language* for the definition of a multi-data source, and MRL, the *Multi-data source Retrieval Language* for data retrieval from a multi-data source. Defining a multi-data source in MDL is quite simple and intuitive: a collective name is assigned to a group of data sources. A collective name simplifies query expression; users specify inter-source conflicts between elements composing the multi-data source and store them into an additional specific data source. MRL extends XQuery in order to

access multiple conflicting data sources through a single query. With MRL, it is easy to smooth out all semantic data differences which often exist in autonomous data sources.

In MDL a multi-data source schema is a collection of data source or multi-source schemas. It can be defined as an XML document, according to the Document Type Definition (DTD) presented in Figure 11.4. In such DTD, `< multisources >` and `< source >` elements refer to a specific multi-data source or a data source, respectively. Attribute `< name >` denotes either the name of a source, of a multi-data source, or of a property in a data source. Attribute `< url >` describes the path to reach a data source. Element `< feature >` represents a property of a data source.

Conflicts between data sources are represented in a specific data source (thus, an XML document) called *Conflicts.xml*. Such data source represents the set M of mappings among data sources, specified to deal with conflicts. The structure of *Conflicts.xml* is detailed in Figure 11.5. Three distinct types of semantic and data conflict information can be represented for elements of a multi-data source: similarity and dissimilarity (elements *Similar* and *Dissimilar*) and scale conflicts (element *Scale*). In all the three cases, the children elements *Node* contain information concerning the elements involved in the declaration. Thus, any two elements children of element *Similar* are considered as semantically equivalent or synonym; any two elements children of element *Dissimilar* are considered as semantically different; any two elements children of element *Scale* are considered semantically similar with conflict of type *scale* (e.g., currency type).

Scale conflicts are resolved through a service. Element *Services* specifies the services (e.g., functions) devoted to resolve a conflict of a specific type (e.g., currency type) available in the multi-data source. A service is selected (during a query's treatment) following the type of conflict that occurs. For example, to resolve a conflict between two nodes N_1 and N_2 under a tag *Scale* with Currency type, a specific service corresponding to this type of conflicts may take node N_1 as input and returns a value that conforms with the currency of the second node N_2 .

Example 2. Figure 11.7 shows the schema of the *Genome* MDS. It is composed of two conflicting data sources SL_1 and SL_2 . They are conflicting since they contain elements with the same name. Conflicts are illustrated in Figure 11.6. File *Conflict.xml* specifies that features (i.e., element) id_1 and *Description* (in SL_1) are respectively similar to id_2 and *Description* (in SL_2).

```
<!ELEMENT multisources (source|multisources)+ >
<!ATTLIST multisources name CDATA #REQUIRED >
<!ELEMENT source (feature)+ >
<!ATTLIST source name CDATA #REQUIRED >
<!ATTLIST source url CDATA #REQUIRED >
<!ELEMENT feature (#PCDATA) >
<!ATTLIST feature name CDATA #REQUIRED >
```

Fig. 11.4 Multi-source DTD.

Element	Children(s)	Attribute(s)	Description
Conflicts	<i>Similar*</i> , <i>Dissimilar*</i> , <i>Scale*</i> , <i>Services*</i>	-	Root element
Similar	<i>Node*</i>	Id	Similar elements. Each similar element has an identifier
Node	Path, Elt, Unit?	-	Description of a Node (i.e., name, path, etc.)
Dissimilar	Path, Elt, <i>Path</i>	Id	Dissimilar elements. Each dissimilar element has an identifier
Scale	<i>Node</i>	Type	Scales conflicts and their types (e.g., currency)
Services	<i>Service*</i>	Type	Available services
Service	<i>Name</i> , <i>Path</i> , <i>Convert</i>	-	Description of a service
Convert	Unit1, Unit2	-	Information used in conversion services
Name	PCDATA	-	Text to be parsed
Unit1	PCDATA	-	Text to be parsed
Unit2	PCDATA	-	Text to be parsed
Path	PCDATA	-	Text to be parsed
Elt	PCDATA	-	Text to be parsed
Unit	PCDATA	-	Text to be parsed

Fig. 11.5 Structure of data source conflicts; labels in italic identify element names.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CONFLICTS SYSTEM "conflicts.dtd">
<CONFLICTS>
  <SIMILAR id="s1">
    <Node>
      <PATH>Bio/Adn/SL1/liste_genes_X/gene</PATH>
      <ELT>id1</ELT>
    </Node>
    <Node>
      <PATH>Bio/Adn/SL2/liste/EnsembleGene_ID</PATH>
      <ELT>id2</ELT>
    </Node>
  </SIMILAR>
  <SIMILAR id="s2">
    <Node>
      <PATH>Bio/Adn/SL1/liste_genes_X/gene</PATH>
      <ELT>description</ELT>
    </Node>
    <Node>
      <PATH>Bio/Adn/SL2/liste/EnsembleGene_ID</PATH>
      <ELT>description</ELT>
    </Node>
  </SIMILAR>
  ...
</CONFLICTS>

```

Fig. 11.6 Conflicts between SL_1 and SL_2 .

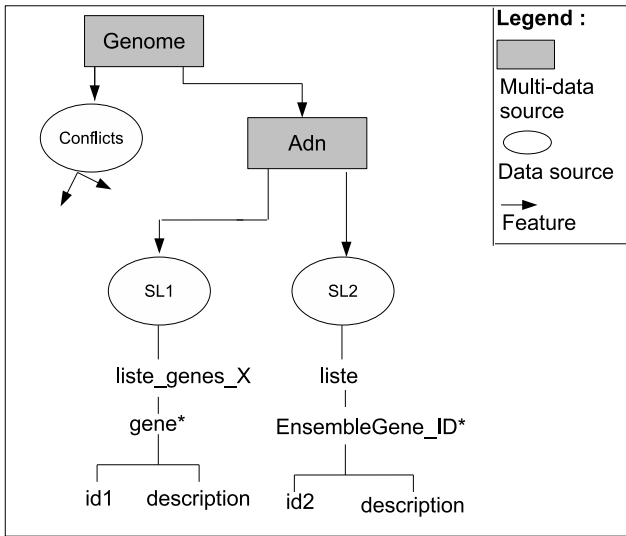


Fig. 11.7 Genome MDS.

11.4.2 Methodology for Semantic Reconciliation

Differently from traditional data integration approaches, like BGLAV, MFA does not require full semantic integration of data sources in order to execute queries on the multi-data source schema. Rather, mappings could be partial or approximated. Hence, MFA offers tools for semantic reconciliation between data sources, i.e., for identifying and solving conflicts between data sources. To this aim, ontologies can be used as part of the integration approach. Alternative approaches should be used whenever no ontology is available for the domain at hand. Examples of alternative methods are: similarity functions between data source elements; methods that infer mappings from answers of queries executed over data sources. The discovered semantic mappings are stored in the data source *Conflicts.xml*, as described in Section 11.4.1 and used later by the query rewriting module in order to answer queries. A user which is unsatisfied of the answers has the opportunity to add (or modify) mappings and submit once again its query. In doing so, the data source *Conflicts.xml* is gradually enriched and the quality of responses becomes increasingly accurate.

Figure 11.8 illustrates a methodology of reconciliation between data sources. The method relies on an ontology for defining concepts, properties, and relationships between these concepts. The usage of an ontology allows the user, at one side, to clearly specify the interest domain and, at the other side, improves the user's knowledge about the data source by clarifying the meaning of all its elements. For example, a user interested in a Biological domain must specify an ontology conformed to this domain. The user can then assign to each element of the data source schema the equivalent semantic element in the specified ontology. This step involves a

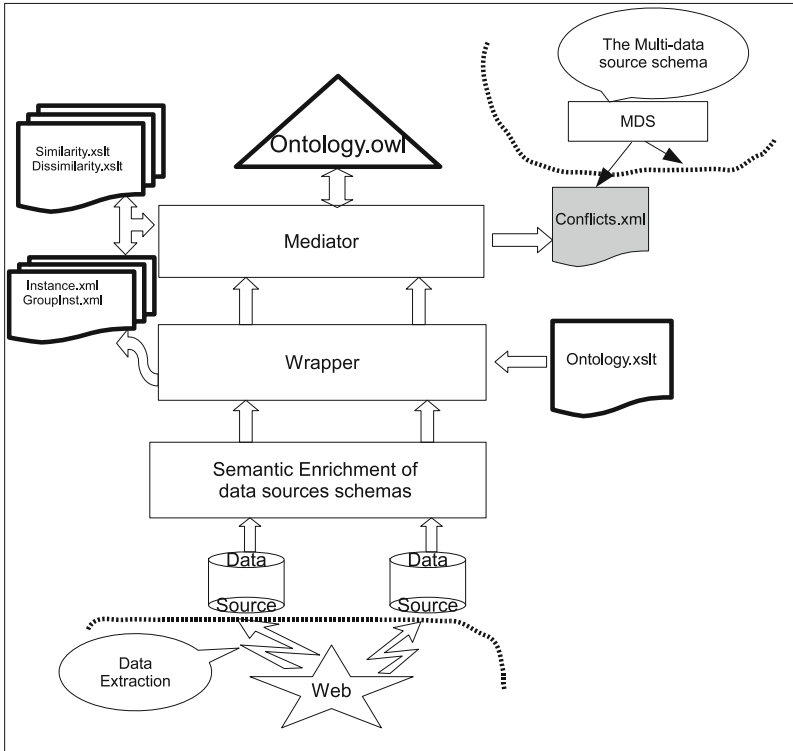


Fig. 11.8 Approach for semantic reconciliation between data sources.

semantic enrichment of the data source schema; such enrichment can be represented as an XML document conformed to the DTD presented in Figure 11.9, called *SemEnr.dtd*.

```

<!ELEMENT source (ontology+) >
<!ATTLIST source name CDATA #REQUIRED >
<!ATTLIST source url CDATA #REQUIRED >
<!ELEMENT ontology (feature) >
<!ATTLIST ontology name CDATA #REQUIRED >
<!ELEMENT feature (ontology?) >
<!ATTLIST feature name CDATA #REQUIRED >

```

Fig. 11.9 Semantic Enrichment DTD.

The next step is to represent data source instances according to the chosen ontology language and the semantically enriched data source schemas generated in the previous step. This conversion is performed by the *wrapper* component, using a defined template (e.g., *Biological.xslt*).

The final step consists in generating the *Conflicts.xml* data source. This activity is performed by the *mediator* component, which has two main tasks:

1. For each data type property, all the instances sharing that property are grouped together. Such step can be accomplished by querying directly the ontology. Assuming the ontology is represented as an OWL/RDF document, ontology queries can be considered at three different levels [17, 4]: syntactic level by using the XQuery language; structure level and semantic level by using an RDF query language such as RQL [20] and SPARQL [31, 17], respectively. A survey and more comparative analysis of different query languages has been published in [13]. The result is stored in an XML document named *GroupInst.xml*.
2. Generate information about similarities (dissimilarities) between data source instances by transforming the document *GroupInst.xml*, using an appropriate template (e.g., *Similarity.xslt* or *Dissimilarity.xslt*). The result of this step is stored in the document *Conflicts.xml*. In this document, a node with a tag *Similar* represents a semantic link between two elements of equivalence or synonym types. A node with a tag *Dissimilar* represents a semantic link between two elements of homonym or disjoint types. Each node, with a tag *Similar* or *Dissimilar*, is associated with an identifier $s_i(d_i)$.

11.4.3 Query Processing in MFA

In this section, we highlight MFA query processing. First we provide some necessary background on the type of the supported queries, then we illustrate the query rewriting steps.

11.4.3.1 Type of Queries in MRL

In MRL, a query is defined as follows:

```
Use (multi-)datasource1 name1 [(multi-)datasourcej namej]*
Allow $ < semantic – variables >
(E)XQuery query
Close name1 [,namej]*
```

Clauses *Use*, *(E)XQuery* and *Close* are mandatory whereas clause *Allow* is optional. Clause *Use* delimits the scope of the query and connects to (multi-)data sources for processing whilst *Close* disconnects from such data sources. *name_j* is a given alias for either a data source or a multi-data source; clause *Allow* is used for the declaration of semantic variables. Through these variables, the user declares his/her intention to access data, in a given query, semantically similar and differently named. An (E)XQuery expression can be formulated as an XQuery query [1] or as an EXQuery query, as defined in [29]. In this last case, active data sources representing processing unit (e.g., web services) can also be invoked.

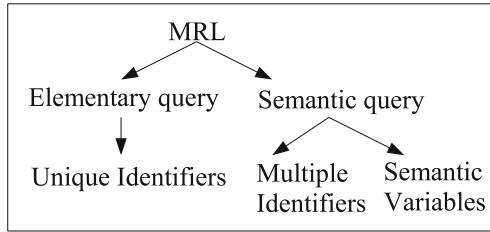


Fig. 11.10 Classification of MRL queries.

MRL queries can be classified depending on the type of paths (also called *designators*) appearing inside the (E)XQuery query component. Indeed, it is important to distinguish queries which access elements whose tag belongs to a single data source (elementary queries) from queries which access elements whose tag belongs to several data sources (*semantic queries*). The overall query classification is presented in Figure 11.10. Elementary and semantic queries can therefore be defined depending on the type of *identifiers* they contain. Elementary queries only contain *unique identifiers*, as defined below.

Definition 5 (Unique identifier). A unique identifier is a designator that univocally identifies an element in the scope of the query.

Semantic queries are used when various data sources represent the same universe in possibly different ways. Semantic queries are also called broadcast queries [24] because a user may have to broadcast the same query to several data sources. In its current form, XQuery does not easily capture such situations: indeed, with XQuery, the user needs to formulate as many queries as there are data sources. In contrast, semantic queries allow to broadcast the user intention in a single query. This is a major simplification, especially for a larger scope. Syntactically, semantic queries are formulated as elementary queries but rely on the usage of *multiple identifiers* and *semantic variables*.

Definition 6 (Multiple identifiers). A multiple identifier is a designator that identifies more than one element in the scope of the query.

Definition 7 (Semantic variables). A semantic variable is a variable whose domain is a set of elements that are semantically similar.

The aim of semantic variables is to enable the user to broadcast his/her intention over different elements which are related by similarity relationships inside the *Conflicts.xml* data source. A semantic query with semantic variables is considered as the set of pertinent elementary sub-queries resulting from all possible substitutions of semantic variables and multiple identifiers by unique identifiers.

Semantic variables can be declared inside the *Allow* clause of a MRL query according to the following syntax:

Allow $\langle \text{semantic-variable} \rangle = \langle \text{designator} \rangle [\langle \text{designator} \rangle]^+$
 $\langle \text{semantic-variable} \rangle ::= \langle \text{simple-variable} \rangle | \langle \text{composed-variable} \rangle$
 $\langle \text{composed-variable} \rangle ::= \langle \text{simple-variable} \rangle [\langle \text{simple-variable} \rangle]^+$
 $\langle \text{simple-variable} \rangle ::= \langle \text{string} \rangle$
 $\langle \text{designator} \rangle ::= \langle \text{string} \rangle [\langle \text{string} \rangle]^+$

We notice that, in MFA, unlike in BGLAV, we are faced with a single type of queries where the tree of the user's query is completely covered by the schemas of data sources. This is due to the fact that while in BGLAV an element E in the global schema is mapped into one or several elements (E_1, E_2, E_i) in the data sources, in MFA each element E in the global schema is mapped onto itself in the corresponding data source.

Example 3. Consider the MDS presented in Example 2. The query Q_1 below extracts the description associated with a gene identifier, posed over the *Genome* MDS.

Q_1 :

```

Use Adn ad
Allow $a = id1.id2
For $x in document('mds')/Genome/ad
where $x/*/ $a='ENSG000001018941' or $x/*/ $a='ENSG00000146950' return
  <Result>
    $x/*/ $a, $x/*/Description
  </Result>

```

In query Q_1 , variable a is a semantic variable whose domain is $\{id_1, id_2\}$. Feature *Description* is a multiple identifier since it designates the *Description* feature in both data sources SL_1 and SL_2 .

Example 4. Figure 11.11 describes a multi-data source named *DNA*, composed of two static data sources *Fragments* and *List_genes*. *Concat* and *Convert* are two active data sources that compose a multi-data source called *Services*. *Conflicts* describes the conflicts between *Fragments* and *List_genes* (not detailed in this figure). *Services*, *Conflicts* and *DNA* constitute a multi-data source called *Biology* which, in this case, is the root of the overall multi-data source. An MRL query is expressed directly over the *Biology* MDS; the answer is the union of answers returned by each component data source, namely S_1 and S_2 . The only problem that arises here is how to solve conflicts between elements belonging to the user's query: Subsection 11.4.3.2 details the solution.

11.4.3.2 Query Rewriting in MFA

In this section, we detail step by step the algorithm for Rewriting Semantic Queries (RSQ) [28]. The overall process involves five steps: (i) query analysis; (ii) creation of the query tree; (iii) searching for semantic ambiguous elements; (iv) generation of sub-queries; (v) query execution. In the following, each step will be discussed in details.

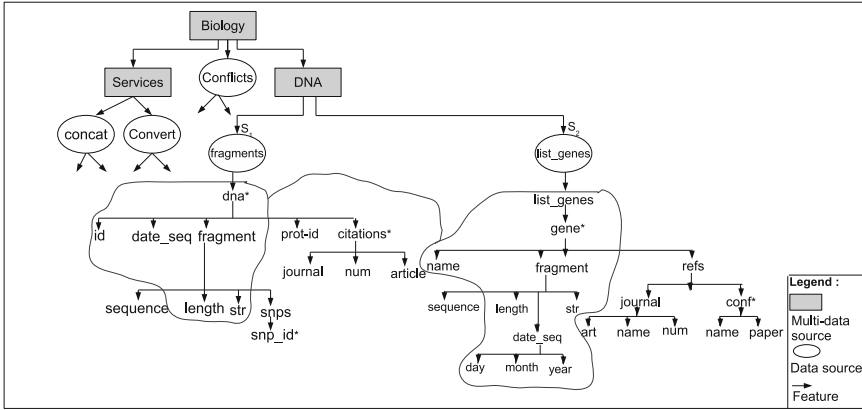


Fig. 11.11 Examples of queries in MFA.

Step 1: Query analysis (see Algorithm 1). This step consists in analyzing clauses *Use*, *Allow*, *For* (or *Let*) and *Return* (the last three appearing in the (E)XQuery query) of an input MRL query. It returns the following tables:

- table *MsoVarTab*, containing the names of data sources or multi-data sources with their related aliases;
- table *SemVarTab*, containing the semantic variables that have been specified in the *Allow* clause and their corresponding definition;
- *ConVarTab*, containing the different context variables specified in clauses *For* or *Let* and their respective values;
- *ResVarTab*, containing the (sub-)set of context variables specified in the clause *Return*.

The outcome of Step 1 for the query presented in Example 3 is illustrated in Figure 11.12.

Step 2: Creation of the query tree (QTree) (see Algorithm 2). Information gathered in Step 1 (*MsoVarTab*, *SemVarTab*, *ConVarTab*, *ResVarTab*) is used to build

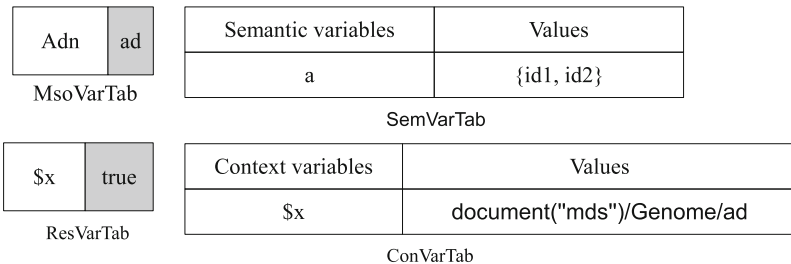


Fig. 11.12 Analysis of the query’s clauses.

Algorithm 1. AnalyseQuery(Q_{MRL}): Analysis of the user query.

Require: **Input:** Q_{MRL} MRL query which is syntactically correct

Output: **MsoVarTab, SemVarTab, ConVarTab and ResVarTab:**

MsoVarTab contains the names of multi-data sources and their alias present in the clause *Use*

MsoVarTab contains the semantic variables present in the clause *Allow*

ConVarTab contains the variables declared in the body of the query

ResVarTab contains the variables present in the clause *Return*

- 1: Initially, these variables are Empty.
 - 2: $MsoVarTab \leftarrow AnalyseClauseUse(Use)$
 - 3: $SemVarTab \leftarrow AnalyseClauseAllow(Allow)$
 - 4: $ConVarTab \leftarrow AnalyseVarOfQuery(Body)$
 - 5: $ResVarTab \leftarrow AnalyseVarOfQuery(Return)$
 - 6: Return $MsoVarTab, SemVarTab, ConVarTab$ and $ResVarTab$
-

the query tree, denoted by QTree. QTree describes the *context* of each element used in the query. The context of an element E is defined as the path that connects the root of the MDS to E . Each node in the QTree is labeled either with the path characterizing a given data source or with the path characterizing an element inside a data source. Paths are generated from those appearing in the query by replacing each variable with the corresponding values, according to the content of the input tables. Each leaf node in QTree is decorated with the two following information: (1) the set of conflict identifiers in *Conflicts.xml* where the element associated with the leaf node appears; (2) the name of the variables in clause *Return* (e.g., \$a) in which the element associated with the leaf node appears. These information are used later in order to generate semantically coherent sub-queries. The result of this step is illustrated in Figure 11.13.

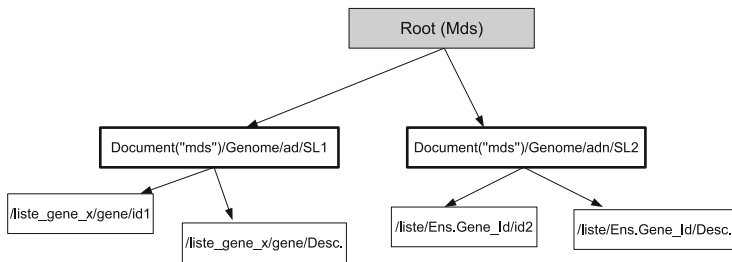


Fig. 11.13 QTree Creation.

Step 3: Searching for Semantically Ambiguous Elements (see Algorithm 3). For the sake of simplicity, we suppose that the query tree is represented through a table called QTab. QTab contains a column for each data source involved in the query and a row for each variable specified in the *Return* clause of the query. The table cell corresponding to a given data source S and a given variable x contains a set of

Algorithm 2. BuildQueryTree(*MsoVarTab, SemVarTab, ConVarTab* and *ResVarTab*): Construction of the tree of the query.

Require: Input: *MsoVarTab, SemVarTab, ConVarTab, ResVarTab*

Output: *QTree*

- 1: $QTree \leftarrow \emptyset$ { Tree of the query }
 - 2: $QTree \leftarrow BuildTree(MsoVarTab, SemVarTab, ConVarTab, ResVarTab)$
 - 3: $QTree \leftarrow EnrichedTree(QTree, Conflicts.xml)$
 - 4: Return *QTree*
-

elements assigned to *x* in *S*. Each element can be associated with either a unique or multiple identifier in the query. The set is empty if no element in *S* is associated with *x* in the query.

Algorithm 3 checks each element in *QTree* as follows: if an element is a multiple identifier for a data source (i.e., it is associated with at least two contexts inside a data source), then this element is considered semantically ambiguous and consequently the *QTree* is ambiguous. In the example illustrated in Fig 11.14, *QTree* is not ambiguous and we move to Step 5.

Algorithm 3. Check(QTree): Search ambiguous elements in *QTree*.

Require: Input: *QTree*

Output: *Boolean*

- 1: $Boolean \leftarrow CheckQTree(QTree)$ { Checks if any element in *QTree* is semantically ambiguous }
 - 2: Return True or False
-

	SL1	SL2
\$x	/liste_gene_x/gene/id1 {s1}	/liste/Ens.Gene_Id/id2 {s1}
	/liste_gene_x/gene/Desc. {s2}	/liste/Ens.Gene_Id/Desc. {s2}

Fig. 11.14 Checking semantic conflicts in *QTab*.

Step 4: SubTrees Generation (see Algorithm 4). This step is invoked only if *QTree* has been considered ambiguous in the previous step. Recall that *QTree* contains all semantic information about the user’s query. This tree allows to check all semantic equivalences between elements and therefore all query conflicts. A *SubTree* is a semantic tree with the same structure of *QTree* but with the following restriction: each return variable and each data source in a *SubTree* is associated with elements (if any) defined within a single context (i.e., unique identifiers). Each *SubTree* leads to a set of pertinent sub-queries, which are semantically coherent.

When QTree is considered ambiguous, it means that one identifier is associated with an element with at least two contexts. In this case at least two semantic trees (SubTrees) are generated. The overall idea is that of separating each context in a sub-tree. Thus, each sub-tree contains only elements of the query which are semantically coherent.

From there, the generation of SubTrees requires the control of conflicts between elements of the QTree. Controlling conflicts between elements consists in checking, for each variable of a user's query (e.g., \$a, given in a row) and for each data source involved in the query (given in column), the elements that have similar identifiers (i.e., the elements designated with the same number s_i means that these elements are semantically similar). The result is stored in the corresponding SubTree. This task is repeated until all cases are processed and the set of SubTrees is generated.

Algorithm 4. *GenerateSubTrees(QTree)*: Generate SubTrees which are semantically coherent.

Require: Input: QTree

Output: at least two trees (SubTree) are generated

```

1: SetOfSubTrees  $\leftarrow \emptyset$ 
   {while remains cases not treated;}
2: while true do
3:   SubTree  $\leftarrow$  GenerateSubTree()
   {Generate empty SubTree having the same structure as QTree}
4:   for all Variable (V)  $\in$  QTree do
5:     SimilarElement  $\leftarrow$  CheckConflicts(V,QTree)
   {for a variable V (e.g., $a) asked by the user in the clause RETURN of the query,
   check conflicts between elements through the set of data sources involved in this
   query, and returns similar elements}
6:     SubTree  $\leftarrow$  UpdateSubTree(V, SimilarElement)
   {Update the SubTree which is semantically consistent}
7:   end for
8:   SetOfSubTrees  $\leftarrow$  SetOfSubTrees  $\cup$  SubTree
9: end while
10: return SetOfSubTrees

```

Step 5: *Generate pertinent sub-queries and Query Execution Plan* (see Algorithm 5). For each SubTree, a set of pertinent sub-queries is generated, which are semantically coherent with each others. Each generated sub-query is an (E)XQuery expression. This step generates ultimately the query plan for each local source.

Notice that, in a general settings, since the number of sub-queries can be very high, after this step the user has three choices:

1. to refine his/her query semantically;
2. to execute the sub-queries which require the maximal number of data sources (and/or a minimum number of missing elements in the sub-queries);

3. by default, to browse the whole set of possible responses. This last choice is not realistic since exploring this set is costly.

Algorithm 5. GenerateQEP(SetOfSubTrees): Generate sub-queries.

Require: Input: SetOfSubTrees

Output: Query Execution Plan (QEP)

- 1: **for all** SubTree \in SetOfSubTree **do**
 - 2: subqueries \leftarrow GenerateSubQueries(SubTree)
 {Generate semantically consistent subqueries}
 - 3: Scheduling(P)
 - 4: Save(P)
 - 5: Return P
 - 6: **end for**
-

Example 5. Query Q_1 presented in Example 3 during Step 5, is decomposed into two sub-queries Q_{11} and Q_{12} , to be executed over data sources SL_1 and SL_2 , respectively.

Q_{11} :

```
For $x in document('SL1')/liste_gene_X/gene
where $x/id1='ENSG000001018941' or $x/id1='ENSG00000146950'
return
  <Result>
    $x/id, $x/description
  </Result>
```

Q_{12} :

```
For $x in document('SL2')/liste/EnsemblGeneID
where $x/id2='ENSG000001018941' or $x/id2='ENSG00000146950'
return
  <Result>
    $x/id2, $x/description
  </Result>
```

11.5 Application

In this section, we consider an application in Genomics and we discuss query rewriting performed according to BGLAV and MFA approaches. In particular, we start by providing the description of the considered data sources, assuming that some restrictions exist concerning data access in each source. Then, for both BGLAV and MFA approaches, we first propose a global schema, i.e., a mediation schema for BGLAV and a multi-data source schema for MFA. We also provide information about conflict management, i.e., a set of correspondence queries for BGLAV and document *Conflicts.xml* for MFA. Finally, we present some queries and we show how they can be rewritten into queries over the data sources.

11.5.1 Data Source Description

Figure 11.15 illustrates three data sources that contain information about the DNA of the human X chromosome. Data are extracted from the well known Ensembl database² then split into different files in order to simulate conflicts such as scale and name conflicts.

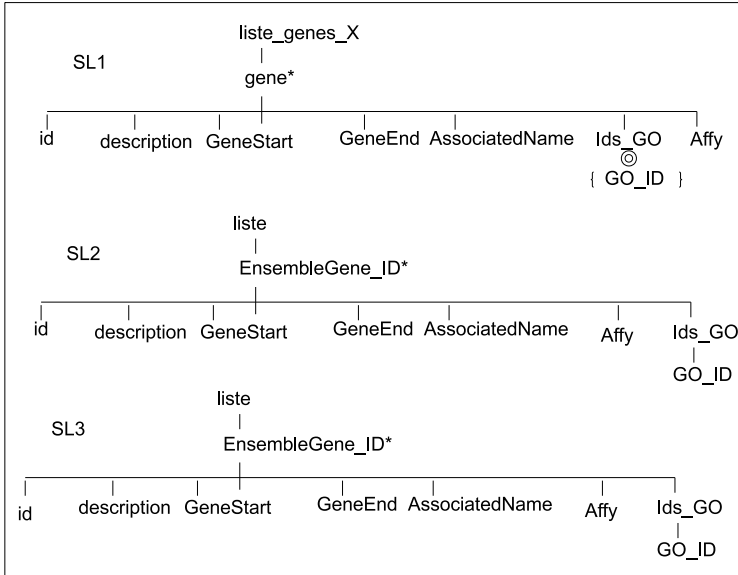


Fig. 11.15 Human chromosomes schemas.

We suppose that there are access restrictions on data sources. In our example, we assume the following constraints:

1. for SL_1 , the value of the feature $[/liste_Genes_X/Gene/ID]$ must be specified in order to access its data;
2. for SL_2 , the value of the feature $[/liste/EnsemblGene_ID/ID]$ must be specified in order to access its data;
3. for SL_3 the value of the feature $[/liste/EnsemblGeneID/GeneStart]$ must be specified in order to access its data.

11.5.2 BGLAV Illustrating Examples

Figure 11.16 shows the global schema and some correspondence queries between the schemas of data sources SL_1 , SL_2 and SL_3 and this global schema.

² www.ensembl.org

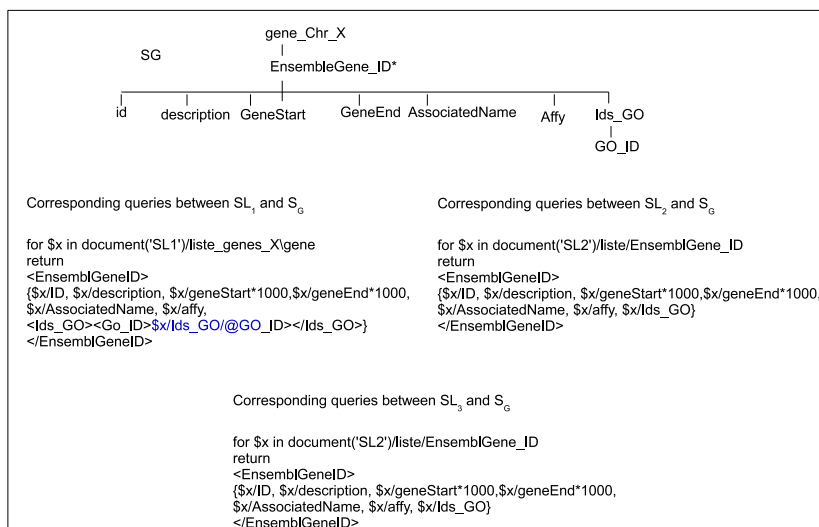


Fig. 11.16 Correspondence queries in BGLAV.

First, let us consider query (Q_2) that extracts values associated with the identifier of a given gene. This query illustrates the concept of query introduced in Definition 3.

Q_2 :

```
For $x in document('S_G')/Gene_Chrr_X/EnsemblGeneID
where $x/ID='ENSG000001018941' or $x/ID='ENSG00000146950'
return
<Result>
  $x/ID, $x/Description, $x/GeneStart, $x/GeneEnd}
</Result>
```

Query Q_2 is decomposed into two sub-queries Q_{21} and Q_{22} targeting respectively SL_1 and SL_2 and presented below. Data source SL_3 is not involved in this query since the restriction access to this source is not satisfied.

Q_{21} :

```
For $x in document('SL1')/liste_genes_X/gene
where $x/ID='ENSG000001018941' or $x/ID='ENSG00000146950'
return
<EnsemblGeneID>
  $x/ID, $x/Description, $x/GeneStart*1000, $x/GeneEnd*1000,
  $x/AssociatedName, $x/Aff
  <Ids_GO>
    <GO_ID> $x/Ids_GO/@GO_ID$ </GO_ID>
  </Ids_GO>
</EnsemblGeneID>
```

Q_{22} :

```
For $x in document('SL_2')/liste/EnsemblGeneID
where $x/ID='ENSG000001018941' or $x/ID='ENSG00000146950'
return
  <EnsemblGeneID>
    $x/ID, $x/Description, $x/GeneStart*1000, $x/GeneEnd*1000,
    $x/AssociatedName, $x/Aff, $x/Ids_GO
  </EnsemblGeneID>
```

Now, lets us consider query Q_3 below which involves data source SL_3 :

Q_3 :

```
For $x in document('$S_{G}$')/Gene_Chrr_X/EnsemblGeneID
where $x/GeneStart > '7770303' or $x/GeneEnd < 9092647
return
  <Result>
    $x/ID, $x/Description, $x/GeneStart, $x/GeneEnd,
    $x/AssociatedName, $x/Affy, $x/Ids_GO
  </Result>
```

Query Q_3 is translated into sub-query Q_{31} posed over data source SL_3 and whose expression is as follows: Notice that, due to access restrictions on SL_1 and SL_2 , no sub-queries are generated.

Q_{31} :

```
For $x in document('SL_3')/liste/EnsemblGene_ID
where $x/GeneStart > 7770303 or $x/GeneEnd < 9092647
return
  <EnsemblGeneID>
    $x/ID, $x/Description, $x/GeneStart*1000, $x/GeneEnd*1000,
    $x/AssociatedName, $x/Affy, $x/Ids_GO
  </EnsemblGeneID>
```

11.5.3 MFA Illustrating Examples

Figure 11.17 shows the Genome MDS composed of the three data sources SL_1 , SL_2 and SL_3 presented in Figure 11.15 while Figure 11.18 shows part of the conflicts. Notice that the node <SCALE> describes the elements which are semantically similar but in addition they also represent scale conflicts between them. For example, the two elements *GeneStart* in the two data sources SL_1 and SL_2 are similar and they represent a scale conflict of type *Measure*.

Consider now query (Q_4) that extracts the values associated with a gene identifier.

Q_4 :

```
Use Adn ad
For $x in document('mds')/Genome/ad
where $x/*/ID='ENSG000001018941' or $x/*/ID='ENSG00000146950'
return
  <Result>
    $x/*/ID, $x/*/Description, $x/*/GeneStart, $x/*/GeneEnd
  </Result>
```

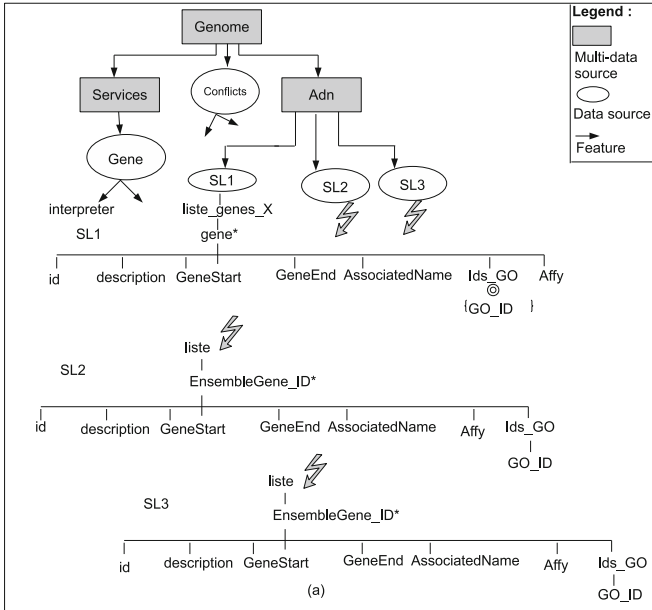


Fig. 11.17 The Genome MDS.

In query Q_4 , id is a multiple identifier since it designates the feature id in the three data sources SL_1 , SL_2 and SL_3 . The same remark applies to features $Description$ and $GeneEnd$. In addition, id , $Description$, and $GeneEnd$ are similar both in the set $\{SL_1, SL_2, SL_3\}$ and the Conflicts data source (see Figure 11.18). Element $GeneStart$ is a multiple identifier in the scope of the query and it represents a scale conflict between the two data sources SL_1 and SL_2 .

Query Q_4 is decomposed into two sub-queries Q_{41} and Q_{42} targeting respectively SL_1 and SL_2 . Data source SL_3 is not involved in this query since the restriction access to this source is not satisfied. Sub-query Q_{41} is an EXQuery expression that involves a static data source and an active one (i.e., the $Gene$ service), while the second sub-query is an XQuery expression.

Q_{41} :

```

For $x in document('SL1')/liste_gene_X/gene
For $y in document('Gene')/service
where $x/ID='ENSG000001018941' or $x/ID='ENSG00000146950'
return
  <Result>
    $x/ID, $x/Description, $x/GeneEnd, $x/AssociatedName, $x/Affy,
    service($y/interpreter, $x/GeneStart), $x/Ids_GO/@GO_ID
  </Result>
    
```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CONFLICTS SYSTEM "conflicts.dtd">
<CONFLICTS>
  ...
<SIMILAR id='s4'>
  <Node>
    <PATH>Bio/Adn/SL1/liste_genes_X/gene</PATH>
    <ELT>GeneStart</ELT>
  </Node>
  <Node>
    <PATH>Bio/Adn/SL2/liste/EnsembleGene_ID</PATH>
    <ELT>GeneStart</ELT>
  </Node>
  <Node>
    <PATH>Bio/Adn/SL3/liste/EnsembleGene_ID</PATH>
    <ELT>GeneStart</ELT>
  </Node>
</SIMILAR>
<SCALE type='Measure'>
  <Node>
    <PATH>Bio/Adn/SL1/liste_genes_X/gene</PATH>
    <ELT>GeneStart</ELT>
  </Node>
  <Node>
    <PATH>Bio/Adn/SL2/liste/EnsembleGene_ID</PATH>
    <ELT>GeneStart</ELT>
  </Node>
</SCALE>
  ...
</CONFLICTS>

```

Fig. 11.18 Excerpt of conflicts in the Genome MDS.

*Q*₄₂:

```

For $x in document('SL_2')/liste/EnsemblGeneID
where $x/ID='ENSG000001018941' or $x/ID='ENSG00000146950'
return
  <Result>
    $x/ID, $x/Description, $x/GeneStart, $x/GeneEnd,
    $x/AssociatedName, $x/Aff, $x/Ids_GO
  </Result>

```

Finally, let us consider query *Q*₅ below:

*Q*₅:

```

Use Adn ad
For $x in document('Multi-Data Source')/Genome/ad
where $x/*/GeneStart > '7770303' or $x/*/GeneEnd < '9092647'
return
  <Result>
    $x/*/ID, $x/*/Description, $x/*/GeneStart, $x/*/GeneEnd,
    $x/*/AssociatedName, $x/*/Affy, $x/*/Ids_GO
  </Result>

```

Query Q_5 is translated into sub-query Q_{51} below and targets data source SL_3 :

Q_{51} :

```
For $x in document('$SL_{3}$')/liste/EnsemblGene_ID
where $x/GeneStart > '7770303' or $x/GeneEnd < '9092647'
return
<Result>
  $x/ID, $x/Description, $x/GeneStart*1000, $x/GeneEnd*1000,
  $x/AssociatedName, $x/Affy, $x/Ids_GO
</Result>
```

11.5.4 Evaluation of MFA Queries

In this section, we study the performance of RSQ algorithms. In particular, we focus on Step 2 of the algorithm, dealing with the generation of the query tree. Recall that this step substitutes each context variable (e.g., \$a) in the input tables with their corresponding values and validates the correctness of their paths on the multi-data source schema. The processing of this step is compared with a baseline version that uses a Cartesian Product (Cart. Prod.) between the set of values taken by the semantic variables in order to find the valid paths on the multi-data source schema.

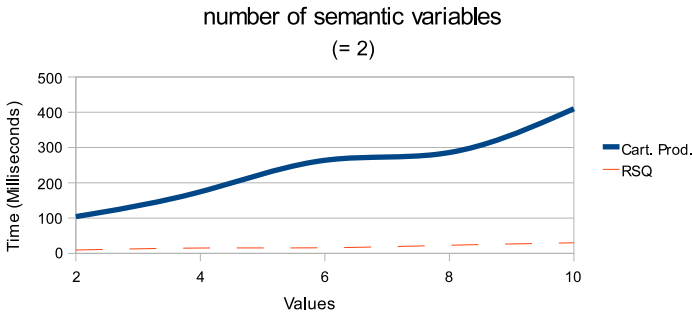


Fig. 11.19 Two semantic variables; each variable takes values varying from 2 to 10.

For each experiment, we considered 20 data sources and we assumed the values taken by each semantic variable vary from 2 to 10. Figure 11.19 shows the case of a query that uses in its body two semantic variables. In this experiment, the response time increases faster in the baseline method, based on 'Cartesian Product', with the increase of the number of values taken by the semantic variables. Figure 11.20 shows the case of a query that uses in its body five semantic variables. In this experiment, the response time in the baseline method increases dramatically, compared to RSQ algorithm, with the increase of the number of values taken by the semantic variables. From the reported experiments, we also observe that the response time is more sensitive to variations concerning the number of semantic variables in a query than to the number of values taken by these variables.

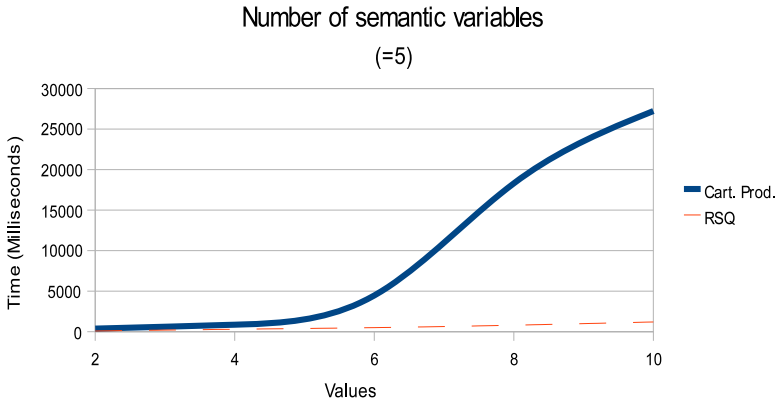


Fig. 11.20 Five semantic variables; each variable takes values varying from 2 to 10.

11.6 Conclusion and Open Issues

In this chapter, we described how to query distributed conflicting web data sources, by means of two data integration approaches. For illustration purposes, we used concrete data drawn from the Genomics domain in a real experimental settings. In order to tackle the data integration problem, we described two approaches: the first one is an XML adaptation of a the well know BGLAV, which pertain to the first-generation of data integration approaches. The second approach, MFA (for multi-source fusion approach) which does not rely on a preexisting mediation schema but rather on a multi-data source schema composed of various data sources, allows flexibility and bootstrapping. Although not being directly inspired by the *dataspace management systems* concepts, MFA relates to this second-generation of data integration approaches.

Because large scale data integration is still a challenge, for future work, we are planning to extend MFA by leveraging existing automated techniques such as schema matching and reference reconciliation: this will help in providing initial correspondences between data sources, hence auto-bootstrapping the system. Feedback from a (more or less) skilled user could be solicited in order to accommodate additional information and build an efficient pay-as-you-go integration system.

References

1. XQuery 1.0: An XML Query Language. <http://www.w3.org/TR/xquery/>
2. ASN.1: Abstract Syntax Notation One, <http://asn1.elibel.tm.fr/en/>
3. Benson, D., Boguski, M., Lipman, D., Ostell, J., GenBank, J.: Nucleic Acids Res. 1–6 (1997)

4. Bönström, V., Hinze, A., Schweppe, H.: Storing RDF as a Graph. In: Proc. of the First Conference on Latin American Web Congress. IEEE Computer Society (2003)
5. Brien, M., Poulouvassilis, A.: Data Integration by Bi-Directional Schema Transformation Rules. In: ICDE, pp. 227–238 (2003)
6. Castano, S., Ferrara, A., Montanelli, S.: H-Match: An Algorithm for Dynamically Matching Ontologies in Peer-based Systems. In: Proc. of the 1st Int. Workshop on Semantic Web and Databases (SWDB) VLDB 2003, pp. 231–250 (2003)
7. Colonna, F.M.: Intégration de Données Hétérogènes et Distribuées sur le Web et Applications à la Biologie. Ph.D. thesis. University Paul Cézanne, Aix-Marseille 3 (2008)
8. Colonna, F.M., Sam, Y., Boucelma, O.: Database Integration for Predisposition Genes Discovery. In: Challenges and Opportunities of Healthgrids, Proc. of 4th HealthGrid Annual Conference. Studies in Health Technology and Informatics, vol. 120. IOS Press (2006)
9. Dong, X.L., Berti-Equille, L., Srivastava, D.: Integrating Conflicting Data: The Role of Source Dependence. In: Proceedings of VLDB 2009, pp. 562–573 (2009)
10. Franklin, M.J., Halevy, A.Y., Maier, D.: From Databases to Dataspaces: a New Abstraction for Information Management. SIGMOD Record 34(4), 27–33 (2005)
11. Friedman, M., Levy, A., Millstein, T.: Navigational Plans for Data Integration. In: Proc. of the National Conference on Artificial Intelligence (1999)
12. Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Vassalos, V., Widom, J.: The TSIMMIS Approach to Mediation: Data Models and Languages. Journal of Intelligent Information Systems 8, 17–132 (1997)
13. Haase, P., Broekstra, J., Eberhart, A., Volz, R.: A Comparison of RDF Query Languages. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 502–517. Springer, Heidelberg (2004)
14. Halevy, A.: Answering Queries using Views: A Survey. Journal of the VLDB, 270–294 (2001)
15. Halevy, A., Franklin, M., Maier, D.: Principles of Dataspace Systems. In: Proc. of PODS, pp. 1–9. ACM Press (2006)
16. Halevy, A., Rajaraman, A., Ordille, J.: Data Integration: The Teenage Years. In: Proceedings of VLDB (2006)
17. Hertel, A., Broekstra, J., Stuckenschmidt, H.: RDF Storage and Retrieval System. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies, pp. 489–508. Springer, Heidelberg (2009)
18. International, R.: The GDB Human Genome Database (2006), <http://www.gdb.org>
19. Jeffery, S., Franklin, M., Halevy, A.: Pay-as-you-go User Feedback for Dataspace Systems. In: Proc. of ACM SIGMOD, pp. 847–859. ACM Press (2008)
20. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A Declarative Query Language for RDF. In: Proc. of the 11th International Conference on World Wide Web, pp. 592–603 (2002)
21. Keen, G., Burton, J., Crowley, G., Dickinson, E., Espinosa-Lujan, A., Franks, E., Harger, C., Manning, M., March, S., McLeod, M., O’Neill, J., Power, A., Pumilia, M., Reinert, R., Rider, D., Rohrllich, J., Schwertfeger, J., Smyth, L., Thayer, N., Troup, C., Fields, C.: The Genome Sequence DataBase (GSDB): Meeting the Challenge of Genomic Sequencing. Nucleic Acids Res. 24, 13–16 (1996)
22. Lenzerini, M.: Data Integration: A Theoretical Perspective. In: PODS, pp. 236–246 (2002)
23. Levy, A., Rajaraman, A., Ordille, J.: Query-Answering Algorithms for Information Agents. In: Proc. of the 13th National Conference on Artificial Intelligence (IAAI 1996), AAAI Press, MIT Press, pp. 40–47 (1996)

24. Lyngbaek, P., McLeod, D.: An Approach to Object Sharing in Distributed Database Systems. In: Proc. of the VLDB, pp. 364–375 (1983)
25. Mootha, V., Lepage, P., Miller, K., Bunkenborg, J., Reich, M., Hjerrild, M., Delmonte, T., Villeneuve, A., Sladek, R., Xu, F., Mitchell, G.A., Morin, C., Mann, M., Hudson, T., Robinson, B., Rioux, J., Lande, E.S.: Identification of a Gene Causing Human Cytochrome Oxidase Deficiency by Integrative Genomics. Proc. of the National Academy of Sciences, 605–610 (2003)
26. Nachouki, G., Quafafou, M.: Multi-Data Source Fusion. Information Fusion 9(4), 523–537 (2008)
27. Nachouki, G., Quafafou, M.: MashUp Web Data Sources and Services based on Semantic Queries. Special Issue: Semantic Integration of Data, Multimedia and Services 36(2), 151–173 (2011); ISSN 0306-4379
28. Nachouki, G., Quafafou, M.: Using Semantic equivalence for MRL Queries Rewriting in Multi-Data Source Fusion System. In: Jin, H. (ed.) Data Management in Semantic Web, pp. 345–382. Nova Science Publishers (2011)
29. Nachouki, G., Quafafou, M., Chastang, M.: A System Based on Multidatasource Approach for Data Integration. In: IEEE-International Conference on Web Intelligence (WI), pp. 438–441 (2005)
30. NCBI: Fasta format. (2006), <http://www.ncbi.nlm.nih.gov/blast/fasta.shtml>
31. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF, W3C Recommendation. (2008), <http://www.w3.org/TR/rdf-sparql-query/>
32. Rahm, E., Bernstein, P.: A Survey of Approaches to Automatic Schema Matching. Journal of the VLDB 10(4), 334–350 (2001)
33. Sarma, A.D., Dong, X., Halevy, A.: Bootstrapping Pay-As-You-Go Data Integration Systems. In: Proc. of ACM SIGMOD, pp. 663–674. ACM Press (2008)
34. Schulze-Kremer, S.: Ontologies for Molecular Biology. In: Proc. of the 3rd Pacific Symposium on Biocomputing, pp. 705–716 (1998)
35. Sheth, A., Larson, J.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Surveys (CSUR), 183–236 (1990)
36. Xu, L., Embley, D.W.: Combining the Best of Global-as-View and Local-as-View for Data Integration. In: ISTA, pp. 123–136 (2004)

Chapter 12

A Functional Model for Dataspace Management Systems

Cornelia Hedeler, Alvaro A.A. Fernandes, Khalid Belhajjame, Lu Mao,
Chenjuan Guo, Norman W. Paton, and Suzanne M. Embury

Abstract. Dataspace management systems (DSMSs) hold the promise of pay-as-you-go data integration. We describe a comprehensive model of DSMS functionality using an algebraic style. We begin by characterizing a dataspace life cycle and highlighting opportunities for both automation and user-driven improvement techniques. Building on the observation that many of the techniques developed in model management are of use in data integration contexts as well, we briefly introduce the model management area and explain how previous work on both data integration and model management needs extending if the full dataspace life cycle is to be supported. We show that many model management operators already enable important functionalities (e.g., the merging of schemas, the composition of mappings, etc.) and formulate these capabilities in an algebraic structure, thereby giving rise to the notion of the core functionality of a DSMS as a many-sorted algebra. Given this view, we show how core tasks in the dataspace life cycle can be enacted by means of algebraic programs. An extended case study illustrates how such algebraic programs capture a challenging, practical scenario.

12.1 Introduction

Given the explosion in the number of data sources that are available for remote access by applications in all areas of activity, it is no surprise that the problem of reconciling the inherent semantic heterogeneity, which such independently designed and maintained sources cannot but exhibit, has grown in importance in the information

Cornelia Hedeler · Alvaro A. A. Fernandes · Khalid Belhajjame · Lu Mao · Chenjuan Guo ·
Norman W. Paton · Suzanne M. Embury
University of Manchester
e-mail: {{chedeler,a.fernandes,khalidb,mao}}@cs.manchester.ac.uk,
{{guoc,norm,sembury}}@cs.manchester.ac.uk

management area. This problem, under the label of *data integration* [5, 21, 31, 45], has been the focus of attention for more than fifteen years. Much progress has been made and several enduring contributions can be discerned such as the idea of mediator-wrapper architectures and techniques for view-based query rewriting that allow a query that is posed against an integration schema to be rewritten as a set of separate queries against the many independent, remote sources it draws data from [29]. Most of these techniques are grounded on two basic capabilities: the first is the ability to perform a semantic matching operation between two sources (typically at both schema and instance levels) that ultimately yields semantic correspondences between them; the second is the ability to derive from these correspondences a semantic mapping, i.e., ultimately, an executable expression that can correctly populate concepts in one schema with instances drawn from concepts in the other.

This technical progress notwithstanding, *data integration systems* (DISs) have only been successful in circumstances where (a) the scale of the required integration is small, (b) the set of data sources (as well as their schemas) is broadly static, and (c) the integration schema has sufficient strategic importance to merit incurring the significant expenditure involved in the expert-intensive matching and mapping stages of the process. As such, and with the benefit of hindsight, one can characterize these achievements as giving rise to integrated resources that have high upfront costs and high quality from the start but whose quality may decay as a result of changes unless high maintenance costs are incurred in the expert-intensive process of propagating the changes. Unfortunately, the continuously growing need for on-the-fly, on-demand combination of data resources (as manifested in the rise of techniques such as web data mash-ups [48], for example, see also Chapter 11) makes this approach (that we might call *traditional, or first-generation, data integration*) fail in terms of cost-effectiveness.

In response to this state of affairs, the idea of *pay-as-you-go data integration* has gained momentum. The realization has grown that users would rather have something that produces results that may be tentative at the start than to have nothing at all (or to have to wait long and pay a lot to have something close to perfect). The vision of *dataspaces* originally proposed by Halevy, Franklin and Maier [27, 30] is that such artefacts will enable agile data integration with much lower upfront and maintenance costs. This *second-generation* approach to data integration is founded, therefore, on the pervasive use of automation in the bootstrapping stages, on the one hand, and on continuous improvement based on user feedback and automated change propagation, on the other hand. The automatic bootstrapping includes the identification of matches and their representation as semantic correspondences and the derivation of mappings. The underlying strategy is based on the idea that if the upfront costs are low, speculative integrations can be attempted among which some will prove useful. Those that do prove useful in principle will thereby motivate users to provide the feedback that, over time, will compensate for the shortcomings associated with the pervasive use of automation in initializing and maintaining the integrated resource.

Among the different conceptions associated with *dataspace management systems* (DSMSs), we have pursued one [6, 7, 33, 34, 35, 36, 50] founded on model management research [2, 9, 10]. Here, we build upon our previous work with a view to describing a comprehensive model of dataspace functionality. We adopt an algebraic style, that is inspired by a tradition of work on database query languages that goes back to E. F. Codd's seminal work and was elegantly continued in the model management area [2, 9, 10]. In particular, we build upon the model management operations in [55, 54]. Our main contribution, therefore, is a formalization of how a dataspace can be operated on, i.e., of a functional model for dataspace, building on a history of advances by data integration and model management researchers whilst giving crisp contours to the notion of a dataspace, contours that had been hitherto only ambiguously and vaguely drawn by other research in dataspace. The hope is that the proposed formalization encourages the use of a consistent terminology in dataspace research and helps consolidate the various research efforts. We do not claim, however, that the operations presented are exhaustive, as we only introduce operations we have been exploring in our previous work. We note, furthermore, that whilst our previous work in dataspace [6, 7, 33, 34, 35, 36, 50] is not incompatible or inconsistent with the model presented here, there are some departures, mostly due to an attempt to be more comprehensive and didactic, whereas, in contrast, our previous work has focussed on more specific aspects or areas of the dataspace literature and has been backed by practical implementation and empirical experimentation.

In our work, we adopt a conception of dataspace as building upon existing research areas. We will make the relationships clearer later but, broadly speaking, we see dataspace as being dependent on automating mapping generation, for which one can use model-management techniques, whose outcome is improved through user feedback. This means that we see model-management techniques primarily as a tool for data integration of existing data resources.

The remainder of the chapter is organized as follows: Section 12.2 introduces the dataspace life cycle. Related work on DSMSs and on model management is discussed in Section 12.3. Section 12.4 introduces the functional model of dataspace as an algebra. Section 12.5 presents examples and use cases to illustrate how the functional model can support the dataspace life cycle and Section 12.6 concludes the chapter and discusses future work.

12.2 Dataspace Life Cycle

It is also useful to consider a dataspace life cycle. This is because a DSMS is a data integration system that relies heavily on automation for bootstrapping and on feedback for improvement, while still having to respond effectively and efficiently to changes to data sources by propagating them through.

We envisage dataspace to have a **life cycle** with the following main stages, or phases: initialization, use, maintenance and improvement [32]. As expected, the initialization stage is a one-off, whereas use, maintenance and improvement phases transition from one to the other until the dataspace is disposed of. We now discuss the stages in slightly more detail.

Initialization: A dataspace must be initialized, or bootstrapped, typically relying on automation whenever past DISs relied heavily on human expertise. In this phase, sources are identified (or discovered), and possibly ranked, before being chosen for participation in the dataspace. Matching techniques [62] can then be used to yield associations between sources that are scored in terms of similarity. Such associations can give rise to semantic correspondences (e.g., of the kind studied in [44], such as, that a concept c in a source s is horizontally partitioned into two concepts c_1 and c_2 in another source s').

The semantic correspondences generated over a set of schemas can then be operated upon using model-management techniques [10]. For example, one can take pairs of semantic correspondences and *compose* them, so that if a construct c_1 corresponds to a construct c_2 , and c_2 corresponds to a construct c_3 , one can obtain from them the semantic correspondence that relates c_1 to c_3 . As another example, the model management technique known as *merge* takes two models (e.g., schemas) along with whatever semantic correspondences hold between them and produces from these a new model that reflects the input correspondences along with two new semantic correspondences that relate the new model to each of the models passed as input. This is, of course, crucial to generate integrated schemas. Note that, unlike in traditional DISs, more than one integrated schema can co-exist in such a dataspace. Semantic correspondences, in turn, can give rise to mappings (e.g., a view that describes c in s as the union of c_1 and c_2 in s'). Once mappings are available, the dataspace has been initialized and is ready for use.

We note that the model-management literature originally focussed on semantic correspondences that are relatively inexpressive. Indeed, [10] seems to be the first paper in that literature to make a case for more expressive semantic correspondences. In our own work, as explained below, we have distinguished between associations, which, as the outcome of matching techniques, are interpretable as claims of similarity; correspondences, which are associations for which there exists evidence that they capture semantic heterogeneity; and mappings, which are correspondences to which one can attach (through algorithmic derivation or through human expertise) an executable expression that reconciles the semantic heterogeneity that the correspondence captures, thereby allowing their use in view-based query evaluation against a mediated schema over autonomous data sources.

Use: A dataspace can be queried using the traditional DIS wrapper-mediator architecture in which a query q against an integrated schema S is rewritten, using the mappings available, in terms of queries q_1, \dots, q_n against sources s_1, \dots, s_n . The results r_1, \dots, r_n from those queries are rewritten, again using mappings, into a result s for the query q against S .

In our own work, we have focussed on query evaluation, but there are many other ways in which a dataspace could be used, e.g., browsing [38], keyword searching [15, 46, 49, 65], or interaction based on the notion of trails [19, 67].

Improvement: This stage is characteristic of dataspace and aims to counteract the shortcomings ensuing from the reliance on automation for bootstrapping (the other characteristic feature of DSMSs in this context). Here, feedback is gathered to which the system response is, under the pay-as-you-go approach, to make the most of the feedback that is given in terms of whatever increase in perceived quality can be obtained from that feedback.

In our own work [6], we have used feedback on query results to annotate, select and refine the collection of mappings that can be used to answer a given query by taking into account the possibilities for trading off precision and recall. We have explored the question of feedback validity and consistency [7]. It is possible and useful to gather other kinds of feedback, e.g., on queries [16]; on mappings [1]; or for query specification [64, 65].

Maintenance: Changes to the sources must be propagated throughout. Thus, changes in source schemas need to be reflected in the mappings, changes in source extents may change the scores that can be assigned to associations, thereby potentially changing which semantic correspondences (and hence mappings) are backed by sufficient empirical evidence.

Figure 12.1 depicts this life cycle in abstract form (using a loosely-notated state-transition diagram, in which ε denotes an automatic transition) while Figure 12.2 zooms into the initialization phase (the other phases being, of course, more application/scenario-specific and, hence, less amenable to being abstracted in the same way).

12.3 Background

This section briefly recalls prior work on DSMSs and on model management. In the case of DSMSs, the picture that emerges is of a field mostly characterized by attempts at defining components (which deliver partial functionality only) or point solutions (i.e., DSMSs that are very specialized either by targeting a specific application domain or by espousing assumptions that significantly narrow the problem scope). In the case of model management, most research has stopped short of tying operations on models to operations on the modelled constructs as supported by traditional *database management systems* (DBMSs). Indeed, proposals for *model management systems* (MMSs) have mainly focussed on the capabilities needed to manipulate and evolve collections of models, as opposed to those that must be available for one to be able to use and improve them. For example, MMSs have often targeted metadata-intensive applications (those in which operations like translating models, manipulating mappings, etc., occur very often) without fully supporting the use of the managed models and mappings (e.g., in query answering, or in extract-transform-load application in data warehousing).

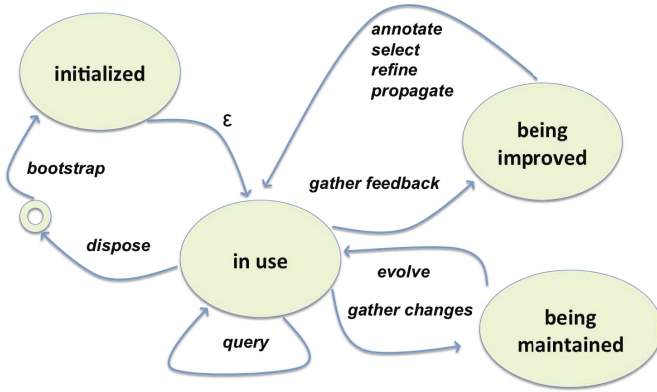


Fig. 12.1 Dataspace: Life cycle.

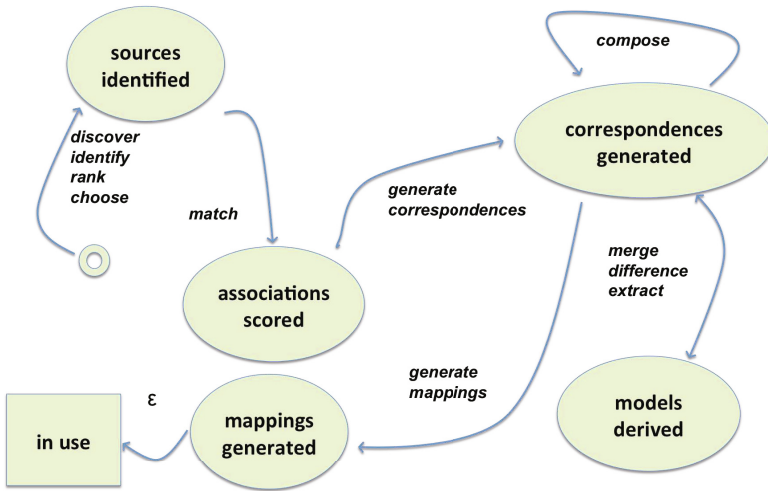


Fig. 12.2 Dataspace life cycle: Initialization stage.

12.3.1 Dataspace Management Systems

We have recently surveyed the literature on dataspace and refer the reader to the detailed findings there [34]. In that work, we devised a classification framework (extending our previous work [32]) in order to characterize and contrast a number of dataspace proposals. Here, we simply delineate the major categories in that framework in order to draw an overall picture of how dataspace are seen by the researchers involved, and, in particular, with respect to what is perceived to be the functionality a DSMS should provide.

DB2 [28] can be seen as a baseline system. It provides queries over heterogeneous sources using a mapping-based mediation approach where mappings are built with significant human intervention, but, significantly, some of the advances resulting from the Clio project [58, 59, 17, 37] were incorporated into DB2. ALADIN [46] supports semi-automatic data integration in the life sciences, with the aim of easing the addition of new data sources. ALADIN is, in this sense, an example of the importance of taking a life cycle view of dataspace. SEMEX [23, 47] integrates personal information. It requires a merged model to be provided upfront and uses it as a pivot to match with and map into the constituent sources. SEMEX responds to changes in sources and to the addition of new sources. iMeMeX [19, 11] also targets personal information, and supports incremental improvement through the manual provision of path-based queries known as iTrails [67]. PayGo [49] aims to integrate web resources. It relies on automation to produce a union schema, uses match to determine how similar constituent schemas are and clusters them. PayGo supports keyword searches but reformulates them into queries that attempt to identify relevant sources using the clusters previously built. UDI [18, 24, 25, 63] is a dataspace proposal for integration of a large number of domain independent data sources automatically. In contrast to the proposals introduced so far, which either start with a manually defined integration schema or use the union of all source schemas as integration schema, UDI aims to derive a merged integration schema automatically, consolidating schema and instance references. In this respect, the conception of a dataspace adopted in UDI is close to the one adopted in our work. Roomba [41] is the first proposal that places a significant emphasis on the improvement phase. It aims to improve the degree of semantic integration by asking users for feedback on matches and mappings between schemas and instances. It addresses the problem of choosing which matches should be confirmed by the user, as it is impossible for a user to confirm all uncertain matches. Matches are chosen based on their utility with respect to a query workload that is provided in advance. ORCHESTRA [65, 39], a collaborative data sharing system, covering the three phases initialization, usage and improvement, uses a generic graph structure to store the schemas and matches between schema elements, which are derived semi-automatically and annotated with costs representing the bias of the system against using the matches. Mappings in the form of query templates are derived from keyword queries posed by the user and matched against the schemas and matches. Cimple [22, 51] aims to reduce the up-front cost of data integration by leveraging user feedback from the community. An integration schema is provided manually, sources matched in a semi-automatic manner in which an automatic tool is used as a starting point and users are asked to answer questions, thus confirming or rejecting matches suggested by the automatic tool. CopyCat [40] follows a more interactive approach to data integration, combining the integration-, usage- and improvement phases by providing a spreadsheet-like workspace in which users copy and paste examples of the data they would like to integrate to answer the queries they have. Similar to CopyCat, OCTOPUS [15] provides the means for integrating multiple sources on the web interactively by providing several operations that can be used to create an integrated data source. Using the SEARCH operator, the user states a keyword query, for which the system tries to find

sources which are ranked according to their relevance with respect to the query. If multiple data sources are required to gather the required information, users can use the `EXTEND` operator, providing a column of a table with which to join the new table and a keyword stating the information desired. With that information the system tries to find appropriate source tables which are ranked according to their relevance with respect to the query and their compatibility with the column provided as input. Throughout the whole integration process, users can provide feedback by editing or annotating in form or rejecting or accepting the suggested source tables. Neither CopyCat nor OCTOPUS distinguish between the various phases of the dataspace lifecycle, e.g., initialization, usage, and improvement. Instead, they promote a seamless combination of initialization, usage and improvement of the dataspace, albeit with a fair amount of user input required.

This brief overview shows that there is broad consensus that dataspace should be construed as second-generation integration platforms in which pervasive automation is deployed to push down costs and user feedback is gathered and responded to increase result quality.

12.3.2 Model Management Systems

The idea of endowing information management systems with the capability to manage models was first formulated in [9] and then revisited in [10]. By *model* is meant, in the information management context, an intensional description of a data resource. By *management* is meant, here, the reification of models as objects of a generic type over which an algebra (i.e., operations on models) is defined.

Model management can be seen as an attempt to simplify the support for a large set of information management tasks that involve operating on models and, in particular, morphisms between models. As listed in [10], such tasks include extract-transform-load in data warehousing; message or data translation; designing portals, forms processors, query interfaces, report writers; and, of particular interest here, wrapper and mediator generation. The most important model-management operations are *match* (which takes models and yields associations or correspondences, i.e., morphisms, between model elements), *merge* (which takes models and correspondences between them and yields a merged model obtained from the input models and correspondences between the former and each of the latter), *extract* (which takes models and correspondences between them and yields the submodel of one of the inputs that can be populated using the input correspondences), *diff* (which takes models and correspondences between them and yields the submodel of one of the inputs that cannot be populated using the input correspondences), and *compose* (which takes two sets of correspondences and yields the correspondences that comprise the composition morphism of the two inputs, seen as morphisms).

The simplification sought by the model management vision is seen to arise from the generic nature of internal representations and the high-level, expressive nature of the generic operations defined on them. The algebraic approach implies compositionality and, for subsets of the algebra, closure. The overall outcome is envisaged

as an expressive algebraic language that can formalize important usage scenarios that would otherwise be quite complex to specify precisely in an error-free manner due to the ad-hoc nature of the representations and the transformations used, as induced by the multiplicity of concrete data modelling formalisms adopted at the user/application level.

The model management area has been surveyed in [9] with respect to previous research that inspired the vision, and in [10] with respect to the proposals that emerged in the wake of [9]. The literature is quite vast and encompasses topics, such as conceptual similarity matching, that, in themselves, have seen voluminous research activity [62].

Broadly construed to encompass systems that are primarily concerned with matching management or with mapping management, as well as the full gamut of model management capabilities, the area has produced impressive research systems such as COMA [20, 4], Clio [58, 59, 17, 37] (some of whose contributions have been incorporated into DB2), AutoMed [13, 12, 61], Rondo [57, 56, 54], GeRoMe [43] and MISM/MIDST [2, 3], among others.

12.4 Functional Model

The purpose of this section is the formulation of a functional model of dataspaces as an algebra that modifies, extends, and complements the core model management techniques in the literature. We make no claim either that different conceptions of dataspaces that emerge from the formulation below are in any way less preferable nor that different formulations of the same conception than the one described below are not possible or as desirable.

We start with the assumption that dataspaces are derived from a collection of existing data resources. We conclude with an algebraic account of the functionalities that our conception of dataspaces makes available. Later, we show how these functionalities can capture interesting, challenging data management scenarios. Firstly, we provide a broad overview of how we construe DSMSs in relation to DBMSs, DISs and MMSs.

12.4.1 An Overview

As already mentioned, we construe DSMSs as combining, adapting and extending the functionality of DBMSs and MMSs. In this respect, we draw the contrast that DISs do not resort to automated matching and mapping generation techniques from MMSs and hence are not as dependent on gathering and responding to user feedback. As broadly suggested by Figs. [12.1] and [12.2], it is possible to provide a clearer picture of the functional relationships between these four classes of systems by building upon and extending previous work in the areas surveyed in Sec. [12.3]. The goal, in this case, is to elucidate, with respect to existing functionality from

DBMSs, DISs, and MMSs, what parts of it can be built upon, what changes and extensions to it are needed, and what additions to it are required.

Figure 12.3 offers a broad overview of how we construe the functional relationships between the four classes of systems. The notation is, loosely, that of data flow diagrams in which arrows denote data (whose types are made clear below), darker boxes denote operations (such as merging or composing models, and annotating or refining mappings) and lighter ovals denote stores (that hold instances of objects such as models, morphisms and feedback). The irregular, dashed-line shapes are used to informally denote the boundaries between the four classes of systems involved thereby roughly indicating what *distinctive* functionality they contribute. We do not mean to imply a strictly component-based architecture by this notational device, e.g., a DIS often comes with its own internal, specialized query evaluation component, and so may a DSMS.

With Figure 12.3 we only aim to note that DSMSs build upon techniques and mechanisms that have been explored in the literature on DBMSs, DISs, and MMSs. Thus, broadly speaking, query evaluation is a core concern in DBMS research, the provision of explicit mappings that give rise to integrated models over existing sources and allow querying over such integrated models to take place lies at the heart of DIS research, and MMS research has focussed from the outset on providing correspondence-driven operations on models by means of which new mappings and models can be derived. In this respect, DSMSs benefit from the research results in those areas and bring a specific concern with improving query results by using feedback to compensate for the shortcomings of the pervasive use of automation instead of intensive reliance on human experts.

The remainder of this section contains our main contribution, viz., an algebraic formulation of the broad functionality depicted in Figure 12.3. Inspiration for this work was drawn primarily from the work on model management, particularly [2, 3, 57, 56, 54]. As we have pointed out, different formulations of the same conception than the one described below (e.g., drawing inspiration from [43] or [13, 12, 61]) are possible and, possibly, as desirable.

12.4.2 Preliminary Assumptions

We assume the existence of a collection \mathbb{D} of data resources. We further assume that a data resource $d \in \mathbb{D}$ can be associated with an intensional description (e.g., a set of statements in some data definition language), as made more precise later, but we abstract over the precise semantics that underpins such intensional descriptions (e.g., the semantics of such modelling notions as inheritance, referential integrity, and similar ones) insofar as our formulation aims to be agnostic as to the data modelling theory or paradigm used for individual data resources.

One of the main purposes of an intensional description is, of course, to characterize the valid states of the corresponding data resource. In other words, from the intensional description of a data resource, one must be able to formally decide whether the data resource is in a valid, or conformant, state at any point in time.

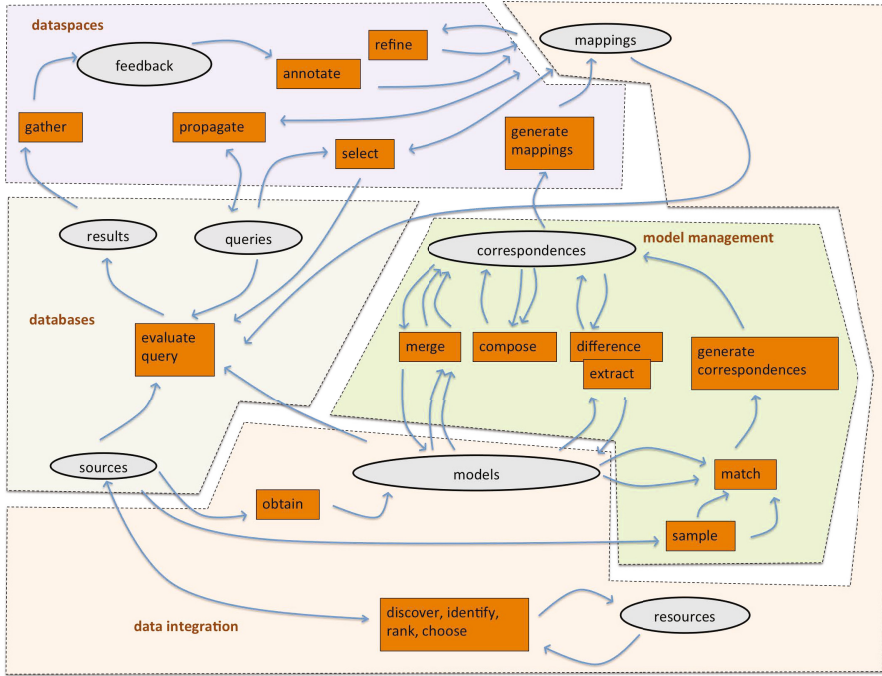


Fig. 12.3 Relating databases, data integration, model management and dataspace.

Here, too, we do not go into the detail of this validity relation, and, instead, assume that the concrete implementations of the many operations on intensional descriptions we characterize later are validity-preserving.

12.4.3 Intensional Descriptions

Let Σ be an infinite set of **symbols** that can act as identifiers of the concepts and values that are modelled in any collection \mathbb{D} of **data resources**.

The intensional description of a data resource $d \in \mathbb{D}$ draws a finite set $C(d) \subset \Sigma$ comprising three pairwise disjoint subsets $SL(d)$, $SA(d)$ and $SR(d)$ containing the symbols used as identifiers of, respectively (and in the terminology of [3]) **superlexicals** (e.g., attributes, in the relational data model), **superabstracts** (e.g., relations) and **superrelationships** (e.g., primary-foreign key pairings).

The intensional description of a data resource $d \in \mathbb{D}$ is, as alluded to above, associated with the set \mathbb{IC} of well-formedness conditions (often referred to as **integrity constraints**) that characterize any state of d as conformant or not to the particular modelling theory or paradigm used to design and deploy it (e.g., in the case of the relational data model, \mathbb{IC} comprises the formalization of notions such as domain integrity, entity integrity, referential integrity, etc.). Since \mathbb{IC} is common to the intensional descriptions of data resources that conform to the data modelling theory

or paradigm from which \mathbb{IC} stems, we omit any reference to it henceforth, thereby assuming that every intensional description is known to be further associated with the set of integrity constraints that is appropriate for it, which, in turn, we assume to be known from the context of the occurrence of the intensional description in the text. In this sense, for the purposes of this chapter, we consider intensional descriptions as syntactic structures over which we define an algebra, as described later.

We use the term **constructs** to refer to superlexicals, superabstracts and superrelationships indistinctly. Thus, given a data resource $d \in \mathbb{D}$, the set of construct identifiers in its intensional description is $C(d) = SL(d) \cup SA(d) \cup SR(d)$.

Let $L(d)$ be a set of **literals** that are possibly specific to a resource d (and hence to the paradigm used to model d). These are used to describe constructs (e.g., in the case of a SQL-based relational data resource, one might find a **schema** named **personnel**, containing, among many others, a **column** of type **varchar** named **address** in a **table** named **employee**, where the terms in **sans-serif** are literals used in that data resource). We assume L to be the union of a set CT of **construct type names** (e.g., **schema**, **table**, **column**), a set CD of **domain names** (e.g., **int**, **varchar**), and a set CN of **construct names** (e.g., **personnel**, **employee**, **name**, **address**, **salary**). Note that construct names and construct identifiers are distinct notions with non-overlapping extensions. Construct identifiers are a notion at the super-(or meta-)model level. They are, therefore, resource-, paradigm- and implementation-independent. Construct names are resource-specific notions (just as construct type names and domain names are paradigm- and implementation-specific ones).

As others have done (most notably [54]) we characterize the intensional descriptions of a data resource d as a graph in which non-leaf nodes are (or are labelled by) construct identifiers in C , leaf nodes are (or are labelled by) literals in $L = CN \cup CT \cup CD$, and edges are subsets of the following relations:¹

$$\begin{aligned} \text{isA} &: C \times CT \\ \text{withDomain} &: C \times CD \\ \text{isNamed} &: C \times CN \\ \text{has} &: C \times C \end{aligned}$$

We use the relation names (i.e., **isA**, **withDomain**, **isNamed**, and **has**) to label the edges of the graph and define, for a data resource $d \in \mathbb{D}$, its **intensional description** (or **model**) to be a graph $G(d)$ with node set $C(d) \cup L(d)$ and edge set $E_1 \subseteq \text{isA} \cup E_2 \subseteq \text{withDomain} \cup E_3 \subseteq \text{isNamed} \cup E_3 \subseteq \text{has}$. Note that, in most practical cases, **withDomain** is more informative for superlexicals. Note also, that one can bind a construct with its extensional description when the data resource is in a given state (indeed this is one way of construing query evaluation, where the query defines a construct whose extension is sought) and represent that with a relation **inState**, though we do not delve into this possibility here.

¹ We abuse notation and omit the reference to a data resource if what we write is valid for all data resources or if the intended reference to specific data resources is clear from context. Thus, we sometimes write C and L rather than the more precise $C(d)$ and $L(d)$, and so on.

Figure 12.4 shows an example intensional description notated as a set of Horn clauses (with an example state of the resource) and Figure 12.5 shows the partial graphical form of the intensional description.

```
% an example intensional description of a resource
% (notated as a set of Horn clauses)

isA(m1, 'schema').
isNamed(m1, 'MGD').
has(m1, r1).
has(m1, r2).

isA(r1, 'relation').          isA(r2, 'relation').
isNamed(r1, 'geneticMarker'). isNamed(r2, 'synonyms').
has(r1, c1).                  has(r2, c4).
has(r1, c2).                  has(r2, c5).
has(r1, c3).

isA(c1, 'column').           isA(c4, 'column').
isNamed(c1, 'accessionID').  isNamed(c4, 'accessionID').
isA(c2, 'column').           isA(c4, 'column').
isNamed(c2, 'chromosome').   isNamed(c4, 'synonym').
isA(c3, 'column').
isNamed(c3, 'symbol').

withDomain(c1, 'varchar').    withDomain(c2, 'integer').
withDomain(c3, 'varchar').    withDomain(c4, 'varchar').
withDomain(c5, 'varchar').
```

Fig. 12.4 Example intensional description.

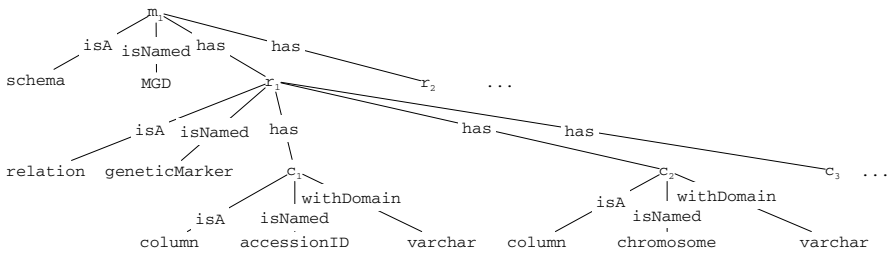


Fig. 12.5 Partial graph representation of example intensional description.

12.4.4 Sorts

Our algebraic specification makes use of several sorts, all of which are alluded to in Figure 12.3 and can be construed as structuring the data from which the operations hinted at in Figure 12.3 and described more formally in Sec. 12.4.5 draw inputs and outputs. For all the sorts described below, we assume that their elements may be described by a **feature set**. For example, an association between ‘worker’ on the one hand and ‘partTimeWorker’ and ‘fullTimeWorker’ on the other typically has a *similarity score* $\in [0, 1]$, which helps distinguish it, when there is an occasion to use that association, from a possibly lower-scored association between ‘worker’, on the one hand, and ‘partTimer’ and ‘fullTimer’, on the other. The former might be reflected in a feature-value pair such as `similarityScore = 0.75` while the latter might have instead `similarityScore = 0.70`. Likewise, a semantic correspondence that postulated that ‘worker’ in schema `s1` is horizontally partitioned into ‘part-time worker’ and ‘full-time worker’ in schema `s2` might have as one of its features the list of selection predicates that decide which tuple in ‘worker’ in `s1` belongs to which partition in the corresponding relations in `s2` (e.g., `selectionPredicateList = [(s2.partTimeWorker, s1.numberOfHours < 8), (s2.fullTimeWorker, s1.numberOfHours ≥ 8)]`). However, we leave the precise, comprehensive and exhaustive definition of such feature sets undiscussed here except where it is crucial for our exposition to consider one or more features.

Table 12.1 summarizes the sorts used in our formulation, which are introduced in the following:

Table 12.1 Sorts.

Sorts	Description
resources	the collection of usable data resources
sources	the subset of resources that have been discovered, identified, ranked or chosen
models	the intensional descriptions of sources
queries	the expressions against a model and a state that, upon evaluation, returns results
results	the instances in the extension of a query returned when the latter is evaluated against a model and a state
associations	the morphisms that relate two sets of constructs by the similarity associated to them by the matching algorithms used
correspondences	the morphisms that, given the matching information, are judged to represent semantic relationships between two sets of constructs
mappings	the morphisms that relates two expressions Q and Q' such that Q computes the extension of Q' and Q' computes the extension of Q when evaluated against their respective models
feedback	the judgements provided by users on the various objects to which they are given access (e.g., query results may be judged for being expected and present, expected and not present, present but unexpected, etc.)

resources. We assume the existence of a collection \mathbb{D} of data resources to each of which an intensional description can be associated. We assume that a data resource is interacted with by means of (G)UI/API mechanisms, i.e., we take it to be an independent software artefact that exposes an interface by means of which people and systems can interact with it.

sources. The set of sources in a dataspace is that subset of the available data resources \mathbb{D} that one has discovered, identified, ranked or chosen to operate on. We assume that the intensional description associated with a data source, as well as a sample of its state, can be obtained. We denote a set of sources with D .

models. In line with most of the background literature, we refer to intensional descriptions (i.e., an instance of a supermodel theory or paradigm) as **models**, as defined above. We denote a set of models with M .

queries. A query is an expression against a model that, upon evaluation, returns a set of results that, as mentioned above, characterizes the extension, at evaluation time, of the construct in the model defined by the query expression. We denote a set of queries with Q .

results. The results returned by query evaluation can be construed as a set of superabstract instances (e.g., tuples in relation ‘employee’), each of which construed, in turn, as a set of pairs, of which the first element is a construct (e.g., a superlexical instance such as the attribute name ‘dept’) and the second denotes the state of that construct (e.g., it is a value, in this case of an attribute, in the domain of the superlexical instance such as ‘Sales’). We denote a set of query results with R .

associations. With or without human intervention, data sources are matched (typically taking account of their models and, sometimes, of samples of their state) in order to produce a similarity score between constructs (e.g., that ‘worker’ in a schema is similar to both ‘partTimeWorker’ and ‘fullTimeWorker’ in another), which sets a feature in the association. We construe an association as a pair of sets of constructs. Thus, we foresee associations as being not just $1 : 1$ but also $1 : n$ and $n : m$ both at superabstract and superlexical levels. It is useful to think of a set of associations as a bidirectional morphism between sets of constructs. We denote a set of associations with A .

correspondences. The associations found through matching can suggest to a human expert or to an algorithm that some semantic relationship (typically equivalence, but also subsumption, and others) is likely to hold. For example, given strong evidence that ‘worker’ in a schema is similar to both ‘partTimeWorker’ and ‘fullTimeWorker’ in another, one could postulate that ‘worker’ in the former is horizontally partitioned, i.e., is the union of ‘partTimeWorker’ and ‘fullTimeWorker’ in the latter, on the basis of a predicate on the attribute ‘numberOfHours’ of ‘worker’. We construe a semantic correspondence as a pair of sets of constructs. Again, it is useful to think of a set of correspondences as a bidirectional² morphism between sets of constructs and we note that, as such, they bear close structural similarity with associations. However, as hinted above, the feature set

² Note that if the relationship is inherently unidirectional, e.g., one of subsumption, then the construal postulated here would have to be refined.

that describes an association is presumed to be different from the feature set that describes a correspondence. This reflects the distinct interpretations we place upon each and, hence, the roles they play in the algebra. An association is presumed to possess very little semantic import (essentially, it postulates a degree of syntactic and structural similarity on the basis of the evidence available to the matching algorithms that were used to derive them). In contrast, a semantic correspondence is presumed to carry significantly more semantic import (e.g., it postulates the existence of a relationship that, more or less directly, points the way to the semantic reconciliation between the related sets of constructs). The additional information that stems from the interpretation we place upon correspondences and that distinguish them from associations and mappings is assumed to be captured in a feature set. We denote a set of semantic correspondences with F .

mappings. The semantic correspondences that are found to be supported by sufficient evidence can be selected by a human expert or by an algorithm for use in mapping a query against a derived, integrated model onto a set of queries against the primary models (i.e., those of the data sources included in the dataspace and involved in the query) and in translating the results emitted by the data source into results structured in terms of the derived, integrated model. We construe a mapping element as a pair of sets of constructs. This means, once more, that mappings (i.e., sets of mapping elements) are bidirectional³ morphisms between sets of constructs and, as such, are structurally similar to sets of associations and to sets of semantic correspondences, with the difference lying, again, on the feature set that describes a mapping element (e.g., as described below, mappings can be annotated for precision and recall on the basis of feedback on the results they produce when used in a query). In this respect, there is one particular characteristic of mappings, viz., that we can, and often do, construe them as views, i.e., executable expressions in a query language. Such an expression can be understood as an intensional description of the mapping (understood, extensionally, as a set of pairs of constructs). As we have shown in [50], given correspondences whose semantic import leads fairly directly to the derivation of view expressions that reconcile some semantic heterogeneities [44], we can algorithmically derive from them the view expression that allows us to populate the source construct(s) with data obtained from the target construct(s) (and *vice versa*), and make it a bound feature in the feature set of a mapping. For example, if there is a semantic correspondence relating ‘worker’ in s_1 to ‘partTimeWorker’ and ‘fullTimeWorker’ in s_2 and if its feature set contains

```
selectionPredicateList = [
    (s2.partTimeWorker, s1.numberOfWorks < 8),
    (s2.fullTimeWorker, s1.numberOfWorks ≥ 8)]
```

then we have shown in [50] how it is possible to derive the view

$$s_1.worker \leftarrow s_2.partTimeWorker \cup s_2.fullTimeWorker$$

³ Note, once more, for inherently unidirectional relationships this construal is too coarse.

that populates the construct in the $s1$ side of the mapping in terms of the constructs in the $s2$ side, as well as the views

$$\begin{aligned} s2.\text{partTimeWorker} &\leftarrow \sigma_{s1.\text{numberOfHours} < 8} s1.\text{worker} \\ s2.\text{fullTimeWorker} &\leftarrow \sigma_{s1.\text{numberOfHours} \geq 8} s1.\text{worker} \end{aligned}$$

that populate the constructs in the $s2$ side of the mapping in terms of the construct in the $s1$ side. In summary, in the case of mappings, we sometimes construe them intensionally as views and sometimes extensionally, as a set of mapping elements (i.e., pairs of sets of constructs). In what follows, we make use of both construals (as exemplified above) without remarking on it unless the context does not suffice to make it clear which one we are using. We denote a set of mapping elements with V .

feedback. The central tenet of a pay-as-you-go approach to dataspace is the recourse to feedback to compensate for the shortcoming of pervasively automating the generation of associations, correspondences and mappings. Feedback can take many forms and an open model of what constitutes feedback is desirable while dataspace research has not yet reached maturity. In our own work [6], we have explored a particular kind of user feedback. More specifically, given the result of evaluating a query that relied on mappings from an integrated schema onto existing sources, if a user provides feedback as to which tuples are true positives (i.e., are in the result and should have been), false positives (i.e., are in the result but should not have been) and false negatives (i.e., are not in the result but should have been), we have shown how such feedback can be used to annotate mappings with estimates of quality (in the form of estimates of precision and recall) that are then used to select which mappings to use in answering queries, as well as to refine mappings (i.e., to derive new mappings from existing ones that are estimated to have better quality than the latter). We denote a set of feedback instances with U .

12.4.5 Operations

This section introduces two groups of operations. The first group acts on elements of the sorts above, or collections formed with such elements. This first group is not characteristic of the functionality exhibited by DBMSs, DISs, MMSs or DSMSs. The second group is, in contrast, characteristic of such systems and constitutes the main focus of interest.

12.4.5.1 Structural Operations

The first group of operations are structural in intention, i.e., they access, transform and derive new elements of the three underlying collection types, viz., sets, graphs, and morphisms. In this sense, they are paradigm- and domain-independent and, mainly, supportive. They were first proposed in [55] and revisited in [54]. Our

account here is closer to the former work in using set comprehension notation for the definition of the operations.

In defining this first group of operations, we use unconventional notation, as follows. Whenever in a signature the type \mathbb{F} occurs, we mean that the signature is valid if every occurrence of \mathbb{F} is simultaneously bound by one of A , F or V (i.e., associations, correspondences or mappings, resp.). Whenever the type \mathbb{S} occurs, the signature is valid if every occurrence of \mathbb{S} is simultaneously bound to either C or L (i.e., construct names or literals, resp.); and, finally, whenever the type \mathbb{X} occurs, the signature is valid if every occurrence of \mathbb{X} is simultaneously bound to the same sort. The signatures of the structural operations, their formal definitions, using set comprehension notation, with short informal descriptions are given in Table 12.2. We note that it is possible to formally define the semantics of these operations more precisely because of their being paradigm- and domain-independent. In contrast, for the second group of operations, their precise definition is more dependent on a modelling paradigm or theory being fixed and made more precise than we can do in the confines of this specific document.

We note that, using the set comprehension notation employed in Table 12.2, we can express the transformation of a set of sets X into the iterative union of its elements as $\text{iUnion}(X) \equiv \{x' | x \leftarrow X, x' \in x\}$. Correspondingly, assuming a set of membership predicates $P = \{p_1, p_2, \dots, p_n\}$, we can express the transformation of a set X into a set of n subsets $\{X_1, X_2, \dots, X_n\}$, where X_i contains those elements for which $p_i \in P$ is true, as $\text{setOfSubsetsOf}(X, P) \equiv \{\{x | x \leftarrow X, p(x)\} | p \in P\}$.

In what follows, the operations in Table 12.2 are not resorted to intensively (though a few definitions in Sec. 12.4.5.5 depend on some of them). This is because the scenarios in Sec. 12.5 can by and large be handled by the operations in Table 12.3. However, those operations are necessary for manipulation of structures (e.g., to convert objects of one sort into objects of another sort). As such, along with syntax for transforming sets into sets of sets and *vice versa*, they are important in making the algebra more effective in modelling less simplified use cases and scenarios than those in Sec. 12.5.

12.4.5.2 Dataspace Operations

The second group of operations is characteristic of the functionality exhibited by DBMSs, DISs, MMSs or DSMSs. Their signatures are given, grouped by system type, in Table 12.3. We construe DISs to subsume the functionality of DBMSs and that of DSMSs to subsume the functionalities of DBMSs, DISs and MMSs. MMSs are seen as providing useful functionality for both DBMSs and DISs and have, therefore, typically been construed as standalone systems with respect to them. As far as we are aware, ours is the first detailed proposal that construes DSMSs as building upon MMS. We note that to assign these operations a precise formal definition is somewhat dependent on a modelling paradigm or theory being fixed (e.g., this is the case for `evalQ`). A semantics for the model-management operations is proposed (and studied for its formal properties) in some detail in [55], which addresses the challenges identified in [9].

Table 12.2 Structural operations.

Operation	Signature	Set comprehension semantics	Description
<i>Primitive</i>			
genId	$\mathbb{S} \rightarrow \mathbb{S}$	$\text{genId}(X) \equiv x \in \Sigma \wedge x \notin X$	Returns a new symbol.
identity	$\mathbb{S} \rightarrow \mathbb{F}$	$\text{identity}(X) \equiv \{(x,x) \mid x \leftarrow X\}$	Returns the identity morphism over the members of X .
domain	$\mathbb{F} \rightarrow \mathbb{S}$	$\text{domain}(X) \equiv \{x \mid (x,y) \leftarrow X\}$	Returns the elements in the domain of morphism X .
invert	$\mathbb{F} \rightarrow \mathbb{F}$	$\text{invert}(X) \equiv \{(y,x) \mid (x,y) \leftarrow X\}$	Returns the inverse of the morphism X .
restrictD	$\mathbb{F} \times \mathbb{S} \rightarrow \mathbb{F}$	$\text{restrictD}(X, X') \equiv \{(x,y) \mid (x,y) \leftarrow X, x \in X'\}$	Returns the morphism whose domain is that of X restricted to those elements that are in X' .
transitive Closure	$\mathbb{F} \rightarrow \mathbb{F}$	$\text{transitiveClosure}(X) \equiv \{(x,y) \mid (x,y) \leftarrow X\} \cup \{(x,z) \mid (x,y) \leftarrow X, (y,z) \leftarrow X\}$	Returns the transitive closure of the morphism X .
constructsIn	$M \rightarrow \mathbb{S}$	$\text{constructsIn}(X) \equiv \{x \mid (x,y) \leftarrow X, x \in C(X)\} \cup \{y \mid (x,y) \leftarrow X, y \in C(X)\}$	Returns all the constructs that occur in the morphism X .
copyUpdate	$M \times \mathbb{S} \rightarrow M$	$\text{copyUpdate}(X, X') \equiv \{(z,y) \mid (x,y) \leftarrow X, x \in X', z \equiv \text{genId}(X)\} \cup \{(x,z') \mid (x,y) \leftarrow X, y \in X', z' \equiv \text{genId}(X \cup \{z\})\}$	Returns a copy of the model X in which those of its elements that occur in X' are given new ids.
submodelOf	$M \times \mathbb{S} \rightarrow M$	$\text{submodelOf}(X, X') \equiv \{(x,y) \mid (x,y) \leftarrow X, x \in X' \vee y \in X'\}$	Returns the submodel of X consisting of its constructs that also occur in X' .
<i>Derived</i>			
range	$\mathbb{F} \rightarrow \mathbb{S}$	$\text{range}(X) \equiv \text{domain}(\text{invert}(X))$	Returns the elements in the range of morphism X .
restrictR	$\mathbb{F} \times \mathbb{S} \rightarrow \mathbb{F}$	$\text{restrictR}(X, X') \equiv \text{invert}(\text{restrictD}(\text{invert}(X), X'))$	Returns the range of the morphism X restricted to those elements that are in X' .
restrict	$\mathbb{F} \times M \times M \rightarrow \mathbb{F}$	$\text{restrict}(X, X', X'') \equiv \text{restrictR}(\text{restrictD}(X, \text{constructsIn}(X')), \text{constructsIn}(X''))$	Returns the morphism whose domain is that of X restricted to members of X' and whose range is that of X restricted to members of X'' .
traverse	$\mathbb{S} \times \mathbb{F} \rightarrow \mathbb{S}$	$\text{traverse}(X, X') \equiv \text{range}(\text{restrictD}(X, X'))$	Returns the range of the morphism returned by restricting the domain of X to elements in X' .
<i>Generic</i>			
union	$\mathbb{X} \times \mathbb{X} \rightarrow \mathbb{X}$	$\text{union}(X, X') \equiv \{x \mid x \leftarrow X\} \cup \{x \mid x \leftarrow X'\}$	Returns the union of two given sets.
minus	$\mathbb{X} \times \mathbb{X} \rightarrow \mathbb{X}$	$\text{minus}(X, X') \equiv \{x \mid x \leftarrow X, x \notin X'\}$	Returns the difference between two sets.
intersection	$\mathbb{X} \times \mathbb{X} \rightarrow \mathbb{X}$	$\text{intersection}(X, X') \equiv \{x \mid x \leftarrow X, x \in X'\}$	Returns the intersection of two sets.

We now comment on the operations whose signatures are given in Table [12.3](#). Again, we group them by system type.

12.4.5.3 Data Source Operations

$r := \text{evalQ}(d, q)$ A query q can be evaluated against a data source d to produce a result r . The modelling theory or paradigm that constrains the representations of superabstracts, superrelationships and superlexicals in d and the syntax and semantics of q define the representation and semantics of the result r . In the classical case of the relational model, d can be construed as a set of tables, q as relational algebraic expression over d and r is characterized by the semantics of the relational algebraic language of which q is an element.

Table 12.3 Dataspace operations.

Operation	Signature	Example invocation	Description
<i>DBMS</i>			
evalQ	$D \times Q \rightarrow R$	$r := \text{evalQ}(d, q)$	Evaluate query q against a data source d to produce a result r .
<i>DIS</i>			
discover	$\mathbb{D} \rightarrow \mathbb{D}$	$D1 := \text{discover}(D)$	Discover data sources.
identify	$\mathbb{D} \rightarrow D$	$D2 := \text{identify}(D1)$	Identify data sources.
rank	$D \rightarrow D$	$D3 := \text{rank}(D2)$	Rank data sources.
choose	$D \rightarrow D$	$D' := \text{choose}(D3)$	Choose data sources.
obtain	$D \rightarrow M$	$m := \text{obtain}(d)$	Obtain model m of data source d .
evalIQ	$Q \times M \times V \rightarrow R$	$r := \text{evalIQ}(q, m, v)$	Evaluate query q over mediated model m using mappings v to produce result r .
<i>MMS</i>			
sample	$D \rightarrow D$	$d' := \text{sample}(d)$	Sample data source d , i.e., get representative content.
match	$D \times D \times M \times M \rightarrow A$	$a := \text{match}(d, d', m, m')$	Match data sources d and d' with models m and m' to obtain set a of associations between them.
infer	$A \rightarrow F$	$f :=$	Generate set f of semantic correspondences
Correspondences		$\text{inferCorrespondences}(a)$	from set a of associations.
compose	$F \times F \rightarrow F$	$f := \text{compose}(f1, f2)$	Compose two sets $f1$ and $f2$ of semantic correspondences into f .
merge	$M \times M \times F \rightarrow M \times F \times F$	$(m, f1, f2) := \text{merge}(m1, m2, f)$	Generate merged model m with corresponding correspondences $f1, f2$ of models $m1$ and $m2$ and set f of semantic correspondences between them.
extract	$M \times F \rightarrow M \times F$	$(m', f') := \text{extract}(m, f)$	Return from model m , given its correspondences f with another model, that portion m' of m that participates in f .
difference	$M \times F \rightarrow M \times F$	$(m', f') := \text{difference}(m, f)$	Return from model m , given its correspondences f with another model, return the correspondences f' of m that does not participate in f .
difference	$F \rightarrow M \times F$	$(f') := \text{difference}(m, f)$	Given model m and its correspondences f with another model, return the correspondences f' that represent the differences between m and the other model.
<i>DSMS</i>			
viewGen	$F \rightarrow V$	$v := \text{viewGen}(f)$	Derive the mappings v from a set f of semantic correspondences.
gather	$R \rightarrow U$	$u := \text{gather}(r)$	Gather set u of feedback instances.
annotate	$U \rightarrow V$	$v := \text{annotate}(u)$	Given set u of feedback instances on results r computed using mapping v annotate v with estimates of its precision and recall.
select	$Q \times V \rightarrow V$	$v' := \text{select}(q, v)$	Given query q and set of mappings v , select set v' of mappings for evaluating q to produce better quality results.
refine	$V \rightarrow V$	$v' := \text{refine}(v)$	Given mappings v , derive new mappings v' from v .
propagateToV	$V \rightarrow V$	$v' := \text{propagateToV}(v)$	Given set of annotated mappings v , use their precision & recall to annotate set of mappings v' with estimated precision & recall.
propagateToQ	$V \times Q \rightarrow Q$	$Q' := \text{propagateToQ}(v, q)$	Given set of annotated mappings v , use their precision and recall to annotate query q with estimated precision & recall of its results.

12.4.5.4 Data Integration Operations

$D1 := \text{discover}(D); D2 := \text{identify}(D1); D3 := \text{rank}(D2); D' := \text{choose}(D3)$

It seems important to distinguish within the set D of available data resources that subset D' of it whose elements are believed to be useful, usable and used. There

are obviously many ways in which the distinction can be drawn on the basis of a different set of operations. Here, for illustration only, we assume that resources are discovered (i.e., ascertained to be useful through mechanisms like search engines, or directory services), identified (i.e., ascertained to be usable, e.g., open to being accessed, in possession of appropriate APIs, sufficiently described, etc.), ranked (i.e., scored with metrics such as authoritative-ness, freshness, completeness, etc.) and chosen (i.e., made members of the dataspace) for actual use. We note that while we seem, above, to suggest that these four operations thread their outputs into inputs, sequentially, this need not be the case.

$m := \text{obtain}(d)$ A model m of a data source d must be obtained so that, as a first class object, it can be scrutinized and manipulated. Under the assumption that the DSMS uses a supermodel theory or paradigm, then m is the outcome of a translation step from the theory or paradigm that constrains d into the DSMS supermodel.

$r := \text{evalIQ}(q, m, v)$ This operation stands in contrast to its single-source form in Sec. [12.4.5.3](#) above. Here, a query q can be evaluated against an integrated set of data D sources to produce a result r , where the integration arises from a set of mappings v over the mediated model m obtained from D . In a global-as-view approach [\[29\]](#), the query q posed against the mediated schema arising from the integration of D is translated using the mapping elements in v over m into single-source queries (see [\[29\]](#) for a survey of the techniques involved and see [\[36\]](#) for an example). Thus, if $D = \{d_1, d_2\}$, then, broadly speaking, two operations are issued, viz., $r_1 := \text{evalQ}(d_1, q_1)$ and $r_2 := \text{evalQ}(d_2, q_2)$ using v over m to derive both q_1 and q_2 from q , and, on the way back, r from r_1 and r_2 .

12.4.5.5 Model Management Operations

$d' := \text{sample}(d)$ Given a data source d , a sample d' of it can be constructed. This operation, among other potential uses, enables a principled reduction (by which we mean one that is representative of the information content of d) in the size of the inputs to the match operation. While many MMSs constrain themselves to matching at schema-level only, additionally matching at instance-level is often a more effective policy if it can be done efficiently, and we construe `sample` as contributing to this purpose. In implementing this operation (and many others below), one expect further parameters (e.g., in the case of `sample`, the desired size of the sample, whether it should be drawn from some given distribution, etc.) but, here, we focus on the abstract operation.

$a := \text{match}(d, d', m, m')$ Given two data sources (or samples thereof) d and d' with their obtained models m and m' respectively, a matching algorithm is used to obtain a set a of associations between them, where each element in a is a pair in which the first and the second element are sets of constructs. Each such pair has in its feature set an assigned `similarityScore` produced by the matching algorithm. There are many ways in which this operation can be implemented [\[62\]](#), the most important dimensions of variation in the present context perhaps being (a) whether the operation uses, internally, multiple matchers making a the

result of an aggregation of similarity scores, possibly independently, produced by distinct matching algorithms; (b) whether both instance- and schema-level information is taken as evidence of similarity or only one of them (typically, schema-level in that case); (c) whether, as suggested above, matching is done pairwise or, alternatively, whether matching is done over a set of data sources taken together, with the former approach being based on an assumption that the associations a_1, a_2, \dots, a_n resulting from a sequence of pairwise matching operations can be aggregated (e.g., by simple union); and (d) whether the associations are, as suggested above, between sets of constructs or between single constructs. In implementing this operation, one might use one or many algorithms, and in the latter case, one might use different strategies for aggregating the scores returned by individual algorithms (see, e.g., [20, 4]).

$f := \text{inferCorrespondences}(a)$ Given a set a of associations, a set f of semantic correspondences can be generated from it. As pointed out in Sec. 12.4.4, semantic correspondences only differ from associations in their information content, i.e., how they are interpretable. It was also pointed out that this distinction can be construed as being captured in a different feature set which annotates associations and correspondences differently. Thus, the former is annotated with, essentially, a similarity score, while the latter are annotated with sufficient information for mappings to be derived that mediate between the sets of constructs involved. Now, invoking $f := \text{inferCorrespondences}(a)$ derives from a subset of the associations in a a set f for each of whose elements, on the basis of matching evidence (i.e., the similarity scores associated with the elements of a), it can be postulated that it can be interpreted as a semantic correspondence. In our own work [50], we have focussed on the semantic correspondences identified in [44] (essentially, same name for distinct constructs, distinct names for the same construct, missing constructs, horizontally- or vertically-partitioned constructs).

$f := \text{compose}(f1, f2)$ Given two sets $f1$ and $f2$ of semantic correspondences, $\text{compose}(f1, f2) \equiv \{(x, z) \mid (x, y) \in f1 \wedge (y, z) \in f2\}$. In other words, when $f1$ and $f2$ are construed as morphisms, in $f := \text{compose}(f1, f2)$ is the composite morphism from $f1$ and $f2$.

$(m, f1, f2) := \text{merge}(m1, m2, f)$ Informally, this operation aims to retain, in the merged model, the information content of the input models. More precisely, given two models $m1$ and $m2$ and the set f of semantic correspondences that are postulated to hold between them, this operation derives a new model m such that $f1$ is a set of correspondences between m and $m1$ and $f2$ is a set of correspondences between m and $m2$, with the following assumption being made in [55]: m is minimal, $\text{constructsIn}(m) = \text{domain}(f1) \cup \text{domain}(f2)$, $\text{range}(f1) = \text{constructsIn}(m1)$, $\text{range}(f2) = \text{constructsIn}(m2)$, and $f = \text{compose}(\text{invert}(f1), f2)$.

$(m', f') := \text{extract}(m, f)$ Informally, this operation aims to return from the input model m , given its correspondences with another model (that may remain unnamed), that portion m' of m that *participates* in f while making the ensuing adjustments in the input correspondences. More precisely, given a model m and a set f of semantic correspondences, this operation derives a model

m' and set of semantic correspondences f' such that [55] m' is minimal, $f = \text{compose}(f', \text{compose}(\text{invert}(f'), f))$, and $\text{constructsIn}(m') = \text{range}(f')$.
 $(m', f') := \text{difference}(m, f)$ Conceptually, this operation acts as the complement of *extract*. Informally, this operation aims to return from the input model m , given its correspondences with another model (that may remain unnamed), that portion m' of m that does *not participate* in f while making the ensuing adjustments in the input correspondences. More precisely, given a model m and a set f of semantic correspondences, this operation derives a model m' and set of semantic correspondences f' such that [55] m' is minimal, $(m', f') = \text{extract}(m, f)$ and $(m, f, f') = \text{merge}(m', \text{compose}(\text{invert}(f'), f'))$.
 $f' := \text{difference}(m, f)$ A variant of the above, this operation, given a model m and its correspondences f with another model, only returns those correspondences f' that express the difference between the input model m and the other model. This variant is the one implemented in [35], but for the purposes of this chapter, we use the version above.

An extended discussion of the properties stemming from the above semantics of *compose*, *merge*, *extract*, and *difference* can be found in [55].

12.4.5.6 Dataspace-Specific Operations

$v := \text{viewGen}(f)$ Given a set f of semantic correspondences, it is possible to derive, algorithmically, the mappings v that the correspondences, conceptually speaking, encode. As mentioned above, in [50] we have shown how *viewGen* can, in fact, generate executable expressions (in our case, *view* expressions against the supermodel we use). In the more general case, the operation is expected to select from the set of semantic correspondences those that are available for use by *evalIQ* in generating queries for *evalIQ*, as described above.

$u := \text{gather}(r)$ This operation is construed as providing the means by which a set u of feedback instances can be gathered. Incremental improvement based on user feedback can take a variety of forms: through the manual provision of mappings (e.g., [67]); through the annotation of query results as to which items are spurious or which should be ranked higher (e.g., [65]); through a more intensively interactive approach requiring a fair amount of user input during the integration process (e.g., [40]), or through a process by which mappings are debugged (e.g., [53]). We observe that all these approaches require, to different degrees, an understanding of the syntax and semantics of mapping and schema languages on the part of the person providing the feedback. This has the drawback that only experts can provide feedback, shutting out casual, non-expert users from the process. This is one reason why, in this formulation, feedback as described in Section 12.4.4 is gathered on query results r (which is the case we have explored in our own work [6]), as this seems to require less expertise and more closely taps into the conceptions and expectations that users have of the dataspace content. The effort in doing this is, so to speak, what the user pays. The operations below illustrate the use that can be made of this kind of feedback and hence the payback for the user in providing it. It is possible, as we have discussed in [33] and [7], to

have different kinds of annotation for which annotation, selection, refinement and propagation would require different algorithms, techniques and strategies which are, for the most part, yet to be pursued in the literature. In [7], we have also discussed approaches to identifying and handling inconsistent feedback.

$v := \text{annotate}(u)$ Given a set u of feedback instances on results r computed using a mapping v , if the instances characterize subsets TP , FP and FN of r , then we can annotate v with estimates of its precision and recall as shown in [6]. The idea in this case is to learn from the feedback given which, among alternative mappings that could be used to answer a query, have the best precision-recall trade-off. Over time, these annotations enable the system to discriminate between mappings that produce good results from those that do not. This is crucial in the dataspace context because of the reliance on automation to derive associations, correspondences and mappings (i.e., the morphisms that give rise to mediated models over independent data sources). The next operations seek to make use of these annotations on mappings.

$v' := \text{select}(q, v)$ Given a query q and set of (presumably annotated) alternative mappings v which could be used for evaluating q , we can select the set v' of mappings that are estimated to produce better quality results (e.g., results to which users assign a better precision-recall trade-off). Our work [6] has provided evidence that with relatively small amounts of effort in the provision of TP, FP, and FN feedback, it is possible to select mappings with a good precision-recall trade-off for a query. However, this assumes that such mappings are already available in the first place, i.e., that the automated generation of mappings succeeded in generating good initial ones. The next operation seeks to cater for the possibility that this may not happen.

$v' := \text{refine}(v)$ Given a set of (presumably annotated) mappings v , we can use their estimated precision and recall to guide a search process over the space of refined mappings, i.e., mappings that are derivable from v , in order to yield a set of mappings v' with better estimates than v . Our work [6] describes the transformations on mappings that generate the space of mappings and an evolutionary algorithm that searches that space. The experimental evidence indicates that this approach is effective in making the most of the feedback (in line with the pay-as-you-go philosophy) to compensate for the potential shortcomings of the pervasive use of automation to bootstrap dataspace.

$v' := \text{propagateToV}(v); q' := \text{propagateToQ}(v, q)$ Given a set of (presumably annotated) mappings v , we can use their estimated precision and recall to annotate a set of mappings v' with estimates of precision and recall, provided that v and v' stand in some relationship (e.g., v' uses v). As a variant operation with a similar intent, given a set of (presumably annotated) mappings v , we can use their estimated precision and recall to annotate a query with an estimate of the precision-recall trade-off that its results would exhibit. These two operations also seek to make the most of feedback. They enable, in the cases described in [6], the propagation of estimates from mappings to mappings and from mappings to queries. This was shown, once again, to compensate, to a cost-effective extent for shortcoming in the bootstrapping phase.

This section has formulated a functional model of dataspace using an algebraic approach that builds on, complements, and extends previous work, particularly in the model management literature. The next section shows how the resulting many-sorted algebra can express frequently occurring use cases and scenarios of great practical interest.

12.5 Bioinformatics Use Case

In this section we illustrate how the functional model in Sec. 12.4 can support the dataspace life cycle using a use case from bioinformatics. Thus, we show in Sec. 12.5.1 how, in our functional model, one can initialize a dataspace and bring it to the point in which it can be used for querying. Then, in Sec. 12.5.2 we show how the functional model supports dataspace maintenance and, in Sec. 12.5.3, dataspace improvement. We illustrate this, firstly, by capturing the consequences of a change in the intensional description of a data source after which the dataspace is, again, ready for querying, and, secondly, by capturing the consequences of the inclusion of additional data sources in the dataspace.

For the purpose of this example, we assume that a group of immunologists is studying malaria. They have decided to use the mouse as a model organism and have carried out a variety of transcriptomics and proteomics experiments, whose results have been deposited in local copies of, resp., Array Express [60], a database of gene expression and other microarray data, and Pride [68], a standards-compliant repository for proteomics data. In order to support the analysis of the data they obtained, various databases need to be integrated. The immunologists choose to bootstrap a dataspace by including in it the following data sources: Genbank [8], which stores the genetic sequences of a large number of organisms; MGD (the Mouse Genome Database) [14], which stores information on mouse including on genetics and on pathways; Gene Ontology [66], which is a knowledge base on the function of genes; and KEGG [42], a pathway database that is conjectured to contain relevant information on pathways additional to those in MGD.

12.5.1 Example: Dataspace Initialization

In this section we present, in Figure 12.6 an example of how a dataspace may come into being by bootstrapping, i.e., by recourse to automation of the matching and merging of data sources and of mapping generation to mediate the interactions between the merged model and that of each data source.

The example can be seen as illustrating how the core functionality of a traditional DIS can be captured in our model of DSMSs. Note that while a traditional approach to integration would be more effective (i.e., deliver high quality results) from start, it would also incur higher costs due to human involvement throughout the set up of

```

1: /* Discover, identify, rank and choose relevant data sources out of a set of bioinformatics data sources
   DS. */
2: {d1, ..., dn} := discover(DS)
3: {d1, ..., dm} := identify({d1, ..., dn})
4: {d1, ..., dm} := rank({d1, ..., dm}) /* with {d1, ..., dm} potentially being in a different order. */
5: /* The data sources chosen are the local copy of ArrayExpress, AEloc; the local copy of Pride, PRloc;
   Genbank, denoted by GB; MGD, and KEGG. */
6: {dAEloc, dPRloc, dGB, dMGD, dKEGG} := choose({d1, ..., dm})
7: D := {dAEloc, dPRloc, dGB, dMGD, dKEGG}
8: /* Obtain the models from the data sources and sample them. */
9: {mAEloc, mPRloc, mGB, mMGD, mKEGG} := obtain(dAEloc) ∪
10:    obtain(dPRloc) ∪ obtain(dGB) ∪ obtain(dMGD) ∪ obtain(dKEGG)
11: {dsAEloc, dsPRloc, dsGB, dsMGD, dsKEGG} := sample(dAEloc) ∪
12:    sample(dPRloc) ∪ sample(dGB) ∪ sample(dMGD) ∪ sample(dKEGG)
13: /* Match the obtained models and generate the correspondences needed to derive merged models. */
14: aAEloc-PRloc := match(dsAEloc, dsPRloc, mAEloc, mPRloc)
15: aAEloc-MGD := match(dsAEloc, dsMGD, mAEloc, mMGD)
16: aGB-MGD := match(dsGB, dsMGD, mGB, mMGD)
17: aMGD-KEGG := match(dsMGD, dsKEGG, mMGD, mKEGG)
18: fAEloc-PRloc := inferCorrespondences(aAEloc-PRloc)
19: fAEloc-MGD := inferCorrespondences(aAEloc-MGD)
20: fGB-MGD := inferCorrespondences(aGB-MGD)
21: fMGD-KEGG := inferCorrespondences(aMGD-KEGG)
22: /* Derive the comprehensive merged model by firstly creating a merged model of the data sources with
   the experimental data sources, then a merged model of Genbank, MGD and KEGG, then merging the
   two resulting models. Then, generate the correspondences between models, composing the correspon-
   dences as needed to map the merged model over all data sources onto the latter. */
23: (mExp, fExp-AEloc, fExp-PRloc) := merge(mAEloc, mPRloc, fAEloc-PRloc)
24: (mGenetic, fGenetic-GB, fGenetic-MGD) := merge(mGB, mMGD, fGB-MGD)
25: fGenetic-KEGG := compose(fGenetic-MGD, fMGD-KEGG)
26: (mGenKEGG, fGenKEGG-Genetic, fGenKEGG-KEGG) := merge(mGenetic, mKEGG, fGenetic-KEGG)
27: fMGD-GenKEGG := compose(invert(fGenetic-MGD), invert(fGenKEGG-Genetic))
28: fExp-GenKEGG := compose(compose(fExp-AEloc, fAEloc-MGD), fMGD-GenKEGG)
29: (mMouse1, fMouse1-Exp, fMouse1-GenKEGG) := merge(mExp, mGenKEGG, fExp-GenKEGG)
30: fMouse1-AEloc := compose(fMouse1-Exp, fExp-AEloc)
31: fMouse1-PRloc := compose(fMouse1-Exp, fExp-PRloc)
32: fMouse1-GB := compose(compose(fMouse1-GenKEGG, fGenKEGG-Genetic), fGenetic-GB)
33: fMouse1-MGD := compose(compose(fMouse1-GenKEGG, fGenKEGG-Genetic), fGenetic-MGD)
34: fMouse1-KEGG := compose(fMouse1-GenKEGG, fGenKEGG-KEGG)
35: /* Generate the mappings needed to evaluate queries against the merged model over the data sources
   in the dataspace. */
36: vMouse1 := viewGen(fMouse1-AEloc) ∪ viewGen(fMouse1-PRloc) ∪
37:    viewGen(fMouse1-GB) ∪ viewGen(fMouse1-MGD) ∪ viewGen(fMouse1-KEGG)
38: /* The bootstrapping is complete. */
39: /* We can now evaluate a query q against the merged model using the generated mappings. */
40: r1 := evalIQ(q, mMouse1, vMouse1)

```

Fig. 12.6 Bioinformatics use case: Bootstrapping a dataspace with Genbank, MGD, KEGG and local copies of ArrayExpress and Pride based on traditional data integration using views to query against a mediated schema over multiple data sources.

the DIS, making this route the slower and more expensive in terms of reaching the stage in which queries can be posed against the mediated model.

Figure 12.6 showing the bootstrapping of this dataspace shows how our functional model captures such tasks as discovering, identifying, ranking (lines 2-4) and choosing the data sources (line 6), then creating the dataspace over the chosen sources by generating a merged integration schema to the point in which it is possible to evaluate a query against this schema that resolves into queries over the sources in the dataspace. To achieve this, the models of the chosen data sources are obtained (lines 9-10) and the data sources sampled (lines 11-12). The models and the samples of the data sources are then used to match them with each other (lines 14-17) and to generate the semantic correspondences (lines 18-21) required to generate a merged schema. Derive the merged model by firstly creating a merged model of the data sources with the experimental data sources (line 23), then a merged model of Genbank, MGD and KEGG (lines 24-26), then merging the two resulting models (lines 27-29), which includes the generation of the semantic correspondences using `compose`. The correspondences between the final merged model and each of the source models are also generated using `compose` (lines 30-34), from which the mappings are generated using `viewGen` (lines 36-37). This completes the bootstrapping process. A query q against the merged model can then be evaluated using the generated mappings (line 40).

Figure 12.6 illustrates how dataspace can support the automated bootstrapping of an artefact which in a traditional data integration would be carefully crafted by human experts.

12.5.2 Example: Dataspace Maintenance

Automated bootstrapping of the kind illustrated in Sec. 12.5.1 can significantly reduce upfront costs. These are not the only high costs incurred in traditional data integration. Another cause for human intervention in traditional data integration scenarios is the need to respond to changes in the data sources, which is particularly problematic when the latter are autonomous (i.e., outside the managerial control of the stakeholder of the integrated resource). There are, among others, the issues of timeliness (i.e., how quickly can changes in the sources be reflected in the models that integrate them) and of cost (insofar as there is a need to resort to human expertise to propagate the changes). This section illustrates how dataspace aim to tackle this issue by automating the propagation of changes in one of the integrated data sources, and the addition of other data sources. The example in Figure 12.7 assumes that the data sources $\{d_{AElloc}, d_{PRloc}, d_{GB}, d_{MGD}, d_{KEGG}\}$ have been integrated as described in Figure 12.6.

For the purpose of the example, we assume that after a period in which the scientists analyse the experimental data using the dataspace, they would like to know how the information gathered and the knowledge obtained transfers across to humans. Therefore, they decide to add two more sources to their dataspace, viz., OMIM (Online Mendelian Inheritance in Man) [52], a database containing

information on human genes and phenotypes, and Ensembl [26], a software system and data resource, that produces, provides and maintains automatic annotation on selected eukaryotic genomes, including both mouse and human. However, they notice that some changes have been made to MGD and decide to update the dataspace first to account for those changes.

The steps involved in updating the dataspace to account for the changes in MGD and those required for adding the two new sources are shown in Figure 12.7. Firstly, the updated model of MGD is obtained and it is sampled (lines 3-4). The new and the old version are matched (line 6), the correspondences over the resulting associations

```

1: /* Assume that  $d_{MGD}$  has changed but we have its previous model  $m_{MGD}$  and a previous sample  $d_{s_{MGD}}$ 
   from it. */
2: /* Obtain the new, changed model of  $d_{MGD}$  and sample the latter again. */
3:  $m_{MGD'} := \text{obtain}(d_{MGD})$ 
4:  $d_{s_{MGD}'} := \text{sample}(d_{MGD})$ 
5: /* Match the new model and the old, generate correspondences over the resulting associations, identify
   the differences between the new model and the old, and, through composition, generate the new
   correspondences needed to derive the new merged model. */
6:  $a_{MGD-MGD'} := \text{match}(d_{s_{MGD}'}, d_{s_{MGD}'}, m_{MGD}, m_{MGD'})$ 
7:  $f_{MGD-MGD'} := \text{inferCorrespondences}(a_{MGD-MGD'})$ 
8:  $(m'_{MGD}, f'_{MGD-MGD'}) := \text{difference}(m_{MGD}, f_{MGD-MGD'})$ 
9:  $f'_{Mouse1-MGD'} := \text{compose}(f_{Mouse1-MGD}, f'_{MGD-MGD'})$ 
10: /* Derive the new merged model from the previous merged model and the changed model  $m_{MGD'}$ ,
   then, through composition, generate the correspondences between the new merged model and the
   unchanged data sources. */
11:  $(m_{Mouse1'}, f_{Mouse1'-Mouse1}, f_{Mouse1'-MGD'}) := \text{merge}(m_{Mouse1}, m'_{MGD'}, f'_{Mouse1-MGD'})$ 
12:  $f_{Mouse1'-AEloc} := \text{compose}(f_{Mouse1'-Mouse1}, f_{Mouse1'-AEloc})$ 
13:  $f_{Mouse1'-PRloc} := \text{compose}(f_{Mouse1'-Mouse1}, f_{Mouse1'-PRloc})$ 
14:  $f_{Mouse1'-GB} := \text{compose}(f_{Mouse1'-Mouse1}, f_{Mouse1'-GB})$ 
15:  $f_{Mouse1'-KEGG} := \text{compose}(f_{Mouse1'-Mouse1}, f_{Mouse1'-KEGG})$ 
16: /* Generate the mappings needed to evaluate queries against the new merged model over the changed
   source(s) in the dataspace. */
17:  $v_{Mouse1'} := \text{viewGen}(f_{Mouse1'-AEloc}) \cup \text{viewGen}(f_{Mouse1'-PRloc}) \cup$ 
18:    $\text{viewGen}(f_{Mouse1'-GB}) \cup \text{viewGen}(f_{Mouse1'-MGD'}) \cup \text{viewGen}(f_{Mouse1'-KEGG})$ 
19: /* This maintenance action is complete. */
20: /* We can now evaluate a query  $q$  against the new merged model using the generated mappings. */
21:  $r := \text{evalIQ}(q, m_{Mouse1'}, v_{Mouse1'})$ 
22: /* Two additional data sources, viz., OMIM and Ensembl, denoted by  $ES$  are to be added to the
   dataspace. */
23: /* Obtain the models from the new data sources and sample them. */
24:  $\{m_{OMIM}, m_{ES}\} := \text{obtain}(d_{OMIM}) \cup \text{obtain}(d_{ES})$ 
25:  $\{d_{s_{OMIM}'}, d_{s_{ES}'}\} := \text{sample}(d_{OMIM}) \cup \text{sample}(d_{ES})$ 
26: /* Match the obtained models with each other and with MGD, which was part of the previous integration
   and contains overlapping information with Ensembl. Then, generate the correspondences needed to
   derive new merged models. */
27:  $a_{OMIM-ES} := \text{match}(d_{s_{OMIM}'}, d_{s_{ES}'}, m_{OMIM}, m_{ES})$ 
28:  $a_{MGD'-ES} := \text{match}(d_{s_{MGD}'}, d_{s_{ES}'}, m_{MGD'}, m_{ES})$ 
29:  $f_{OMIM-ES} := \text{inferCorrespondences}(a_{OMIM-ES})$ 
30:  $f_{MGD'-ES} := \text{inferCorrespondences}(a_{MGD'-ES})$ 

```

Fig. 12.7 Bioinformatics use case: Maintenance by evolving the mediated schema in response to changes in MGD and the addition of OMIM and Ensembl to the dataspace.

```

31: /* Derive the new merged model to include the incoming models, then, through composition, generate
    the correspondences between the new merged model and the preexisting data sources. */
32: (mHuman1, fHuman1-OMIM, fHuman1-ES) := merge(mOMIM, mES, fOMIM-ES)
33: fMouse1'-Human1 := compose(compose(fMouse1'-MGD', fMGD'-ES), invert(fHuman1-ES))
34: (mMH1, fMH1-Mouse1', fMH1-Human1) := merge(mMouse1', mHuman1, fMouse1'-Human1)
35: fMH1-AEloc := compose(fMH1-Mouse1', fMouse1'-AEloc)
36: fMH1-PRloc := compose(fMH1-Mouse1', fMouse1'-PRloc)
37: fMH1-GB := compose(fMH1-Mouse1', fMouse1'-GB)
38: fMH1-MGD' := compose(fMH1-Mouse1', fMouse1'-MGD')
39: fMH1-KEGG := compose(fMH1-Mouse1', fMouse1'-KEGG)
40: fMH1-OMIM := compose(fMH1-Human1, fHuman1-OMIM)
41: fMH1-ES := compose(fMH1-Human1, fHuman1-ES)
42: /* Generate the mappings needed to evaluate queries against the new merged model over the enlarged
    collection of sources in the dataspace. */
43: vMH1 := viewGen(fMH1-AEloc) ∪ viewGen(fMH1-PRloc) ∪ viewGen(fMH1-GB) ∪
44:         viewGen(fMH1-MGD') ∪ viewGen(fMH1-KEGG) ∪ viewGen(fMH1-OMIM) ∪
45:         viewGen(fMH1-ES)
46: /* This maintenance action is complete. */
47: /* We can now evaluate a query q against the new merged model using the generated mappings. */
48: r1 := evalIQ(q, MH1, vMH1)

```

Fig. 12.7 (continued)

generated (line 7), the differences between the new model and the old identified (line 8), and, through composition, the new correspondences generated (line 9) that are needed to derive the new merged model. The new merged model is generated from the previous merged model and the changed model $m_{MGD'}$ (line 11), and then, through composition, the correspondences between the new merged model and the unchanged data sources are generated (lines 12-15). The mappings are generated that are needed to evaluate queries against the new merged model over the changed source(s) in the dataspace (lines 17-18) and queries can now be evaluated (line 21).

To add the two new data sources, their models are obtained and they are sampled (lines 24-25). The obtained models are matched with each other and with the updated model of MGD (lines 27-28), which contains overlapping information with Ensembl, and the correspondences are generated (lines 29-30) that are needed to derive new merged models. Derive the new merged model to include the incoming models (line 32-34), and then, through composition, generate the correspondences between the new merged model and the preexisting data sources (lines 35-41). The mappings needed to evaluate queries against the new merged model are generated over the enlarged collection of sources in the dataspace (line 43-45) and the updated dataspace is ready for queries to be evaluated (line 48).

The scientists then decide that they would like to study some additional experimental data made public by other researchers working on malaria in mouse as well as in human. This requires the integration of the public repositories of ArrayExpress and Pride. As the schemas of ArrayExpress and Pride used for bootstrapping in Figure 12.6 may have evolved since the scientists installed their own local copies, the schemas need to be compared first and differences identified.

The steps involved in integrating the public versions of ArrayExpress and Pride in the dataspace and generating merged schemas for both mouse and the study across mouse and human are shown in Figure 12.8

12.5.3 Example: Dataspace Improvement

We note that, largely relying on the model-management operations, this conception of dataspace provides the basis for supporting more than mediated schema, e.g.,

- 1: /* Two additional data sources, viz., the public versions of ArrayExpress, denoted by AE_{pub} , and of Pride, denoted by PR_{pub} are added to the dataspace. */
- 2: /* Obtain the models from the new data sources and sample them. */
- 3: $\{m_{AE_{pub}}, m_{PR_{pub}}\} := \text{obtain}(d_{AE_{pub}}) \cup \text{obtain}(d_{PR_{pub}})$
- 4: $\{d_{s_{AE_{pub}}}, d_{s_{PR_{pub}}}\} := \text{sample}(d_{AE_{pub}}) \cup \text{sample}(d_{PR_{pub}})$
- 5: /* Match the models of the public versions of ArrayExpress and Pride with the local versions, generate the correspondences and identify the differences, and, through composition, generate the correspondences needed to create a new merged model for mouse and another one for mouse and human. */
- 6: $a_{AE_{loc}-AE_{pub}} := \text{match}(d_{s_{AE_{loc}}}, d_{s_{AE_{pub}}}, m_{AE_{loc}}, m_{AE_{pub}})$
- 7: $a_{PR_{loc}-PR_{pub}} := \text{match}(d_{s_{PR_{loc}}}, d_{s_{PR_{pub}}}, m_{PR_{loc}}, m_{PR_{pub}})$
- 8: $f_{AE_{loc}-AE_{pub}} := \text{inferCorrespondences}(a_{AE_{loc}-AE_{pub}})$
- 9: $f_{PR_{loc}-PR_{pub}} := \text{inferCorrespondences}(a_{PR_{loc}-PR_{pub}})$
- 10: $(m_{AE_{pub}LocDiff}, f_{AE_{pub}-AE_{loc}-Diff}) := \text{difference}(m_{AE_{pub}}, \text{invert}(f_{AE_{loc}-AE_{pub}}))$
- 11: $(m_{PR_{pub}LocDiff}, f_{PR_{pub}-AE_{loc}-Diff}) := \text{difference}(m_{PR_{pub}}, \text{invert}(f_{PR_{loc}-PR_{pub}}))$
- 12: $f_{Mouse1'-AE_{pub}LocDiff} := \text{compose}(f_{Mouse1'-AE_{loc}}, \text{invert}(f_{AE_{pub}-AE_{loc}-Diff}))$
- 13: $f_{Mouse1'-PR_{pub}LocDiff} := \text{compose}(f_{Mouse1'-PR_{loc}}, \text{invert}(f_{PR_{pub}-PR_{loc}-Diff}))$
- 14: $f_{MH1-AE_{pub}LocDiff} := \text{compose}(f_{MH1-AE_{loc}}, \text{invert}(f_{AE_{pub}-AE_{loc}-Diff}))$
- 15: $f_{MH1-PR_{pub}LocDiff} := \text{compose}(f_{MH1-PR_{loc}}, \text{invert}(f_{PR_{pub}-PR_{loc}-Diff}))$
- 16: /* Derive the new merged models for both mouse and the study across mouse and human from the previously merged models and the parts of the public versions of ArrayExpress and Pride that are different from the local versions and generate the correspondences between the new merged models and the integrated data sources. */
- 17: $(m_{Mouse1.5}, f_{Mouse1.5-Mouse1'}, f_{Mouse1.5-AE_{pub}LocDiff}) := \text{merge}(m_{Mouse1'}, m_{AE_{pub}LocDiff}, f_{Mouse1'-AE_{pub}LocDiff})$
- 18: $f_{Mouse1.5-PR_{pub}LocDiff} := \text{compose}(f_{Mouse1.5-Mouse1'}, f_{Mouse1'-PR_{pub}LocDiff})$
- 19: $(m_{Mouse2}, f_{Mouse2-Mouse1.5}, f_{Mouse2-PR_{pub}LocDiff}) := \text{merge}(m_{Mouse1.5}, m_{PR_{pub}LocDiff}, f_{Mouse1.5-PR_{pub}LocDiff})$
- 20: $f_{Mouse2-Mouse1'} := \text{compose}(f_{Mouse2-Mouse1.5}, f_{Mouse1.5-Mouse1'})$
- 21: $f_{Mouse2-AE_{loc}} := \text{compose}(f_{Mouse2-Mouse1'}, f_{Mouse1'-AE_{loc}})$
- 22: $f_{Mouse2-PR_{loc}} := \text{compose}(f_{Mouse2-Mouse1'}, f_{Mouse1'-PR_{loc}})$
- 23: $f_{Mouse2-GB} := \text{compose}(f_{Mouse2-Mouse1'}, f_{Mouse1'-GB})$
- 24: $f_{Mouse2-MGD'} := \text{compose}(f_{Mouse2-Mouse1'}, f_{Mouse1'-MGD'})$
- 25: $f_{Mouse2-KEGG} := \text{compose}(f_{Mouse2-Mouse1'}, f_{Mouse1'-KEGG})$
- 26: $f_{Mouse2-AE_{pub}LocDiff} := \text{compose}(f_{Mouse2-Mouse1.5}, f_{Mouse1.5-AE_{pub}LocDiff})$
- 27: $(m_{MH1.5}, f_{MH1.5-MH1}, f_{MH1.5-AE_{pub}LocDiff}) :=$
- 28: $\text{merge}(m_{MH1}, m_{AE_{pub}LocDiff}, f_{MH1-AE_{pub}LocDiff})$
- 29: $f_{MH1.5-PR_{pub}LocDiff} := \text{compose}(f_{MH1.5-MH1}, f_{MH1-PR_{pub}LocDiff})$
- 30: $(m_{MH2}, f_{MH2-MH1.5}, f_{MH2-PR_{pub}LocDiff}) :=$
- 31: $\text{merge}(m_{MH1.5}, m_{PR_{pub}LocDiff}, f_{MH1.5-PR_{pub}LocDiff})$
- 32: $f_{MH2-MH1} := \text{compose}(f_{MH2-MH1.5}, f_{MH1.5-MH1})$
- 33: $f_{MH2-AE_{loc}} := \text{compose}(f_{MH2-MH1}, f_{MH1-AE_{loc}})$
- 34: $f_{MH2-PR_{loc}} := \text{compose}(f_{MH2-MH1}, f_{MH1-PR_{loc}})$

Fig. 12.8 Bioinformatics use case: Adding the public versions of ArrayExpress and Pride.

```

35:  $f_{MH2-GB} := \text{compose}(f_{MH2-MH1}, f_{MH1-GB})$ 
36:  $f_{MH2-MGD'} := \text{compose}(f_{MH2-MH1}, f_{MH1-MGD'})$ 
37:  $f_{MH2-KEGG} := \text{compose}(f_{MH2-MH1}, f_{MH1-KEGG})$ 
38:  $f_{MH2-OMIM} := \text{compose}(f_{MH2-MH1}, f_{MH1-OMIM})$ 
39:  $f_{MH2-ES} := \text{compose}(f_{MH2-MH1}, f_{MH1-ES})$ 
40:  $f_{MH2-AEpubLocDiff} := \text{compose}(f_{MH2-MH1.5}, f_{MH1.5-AEpubLocDiff})$ 
41: /* Generate the mappings needed to evaluate queries against the new merged models over the enlarged
    collection of sources in the dataspace. */
42:  $v_{Mouse2} := \text{viewGen}(f_{Mouse2-AEloc}) \cup \text{viewGen}(f_{Mouse2-PRloc}) \cup$ 
43:    $\text{viewGen}(f_{Mouse2-GB}) \cup \text{viewGen}(f_{Mouse2-MGD'}) \cup$ 
44:    $\text{viewGen}(f_{Mouse2-KEGG}) \cup \text{viewGen}(f_{Mouse2-AEloc} \cup f_{Mouse2-AEpubLocDiff}) \cup$ 
45:    $\text{viewGen}(f_{Mouse2-PRloc} \cup f_{Mouse2-PRpubLocDiff})$ 
46:  $v_{MH2} := \text{viewGen}(f_{MH2-AEloc}) \cup \text{viewGen}(f_{MH2-PRloc}) \cup \text{viewGen}(f_{MH2-GB}) \cup$ 
47:    $\text{viewGen}(f_{MH2-MGD'}) \cup \text{viewGen}(f_{MH2-KEGG}) \cup \text{viewGen}(f_{MH2-OMIM}) \cup$ 
48:    $\text{viewGen}(f_{MH2-ES}) \cup \text{viewGen}(f_{MH2-AEloc} \cup f_{MH2-AEpubLocDiff}) \cup$ 
49:    $\text{viewGen}(f_{MH2-PRloc} \cup f_{MH2-PRpubLocDiff})$ 
50: /* This maintenance action is complete. */
51: /* We can now evaluate a query  $q$  against the new merged model using the generated mappings. */
52:  $r2 := \text{evalIQ}(q, MH2, v_{MH2})$ 

```

Fig. 12.8 (continued)

alternative ones, and, in the scope of each mediated schema, the coexistence of alternative mappings into the data sources. This pervasive use of automation and this support for alternative models and for alternative mappings for a given model is likely to result in results of lesser quality in comparison to traditional data integration. This section illustrates how dataspace aim to tackle this issue by procuring and responding to feedback. It is assumed that the example in Figure 12.9 uses the dataspace created in the example in Figure 12.8.

For the purpose of the example, we assume that running a set of initial queries shows that the results contain tuples that the immunologists working on mouse and human are not interested in. In particular, since Genbank and KEGG contain information not just on mouse and human but on a variety of other organisms, the queries tend to return tuples containing information on those other organisms too.

The steps for improvement are shown in Figure 12.9, where, for simplicity, the improvement is applied to one query only, and are described in the following. The scientists decide to spend some effort on improving the initial integration by re-running a set of queries and providing feedback on the results, indicating which result tuples they expect to see and which they do not expect to see in the result. The feedback provided is gathered (line 3) and then used to annotate the mappings that were used to evaluate the queries (line 4). This underpins the selection of the mappings to be used in future query evaluations (line 5) and has effects of increasing the likelihood that mappings that have lower quality (e.g., worse precision-recall trade-offs) are excluded.

However, another run of the queries with the new set of mappings (line 10) shows that more improvements can possibly be made. The scientists provide more feedback (line 14) and try refining the mappings (line 16) with the aim of obtaining new mappings that better reflect the expectations of the immunologists.


```

1: /* Assume, following on from Figure 12.8 that some alternative mappings to those initially generated
   have been made available. For the purposes of this example, we assume that alternative mappings
   use  $\mu$  rather than  $\nu$  in the naming scheme we are using. We now wish to pursue opportunities for
   improvement. */
2: /* We start by gathering user feedback on the query result  $r_2$  in Figure 12.8 then using it to annotate
   the available mappings and to select those to be used in the next evaluation of queries (and possibly
   future ones). */
3:  $u := \text{gather}(r_2)$ 
4:  $\{v_{MH2}^a, \mu_{MH2}^a\} := \text{annotate}(u)$ 
5:  $\{v_{MH2}^a, \mu_{MH2}^a\} := \text{select}(q, \{v_{MH2}^a, \mu_{MH2}^a\})$ 
6: /* Note that some of the alternative mapping  $\mu_{MH2}^a$  could be selected over some of the initial mappings
    $\nu_{MH2}^a$  on the grounds that they have been annotated with better quality estimates. */
7: /* For the purposes of informing users (for one example), we can also propagate the quality estimates
   for the selected, annotated mappings to any query that uses them. */
8:  $q^a := \text{propagateToQ}(\{v_{MH2}^a, \mu_{MH2}^a\}, q)$ 
9: /* Now, when we reevaluate the query  $q$ , we can use the selected mappings. */
10:  $r_3 := \text{evalIQ}(q, m_{MH2}, \{v_{MH2}^a, \mu_{MH2}^a\})$ 
11: /* These gather-annotate-select steps can be repeated as many times as necessary in order to select
   the best set of mappings. */
12: /* Assume we still want to pursue further opportunities for improvement. */
13: /* We can gather more user feedback but now on the latest query result  $r_3$ . */
14:  $u_2 := \text{gather}(r_3)$ 
15: /* We can then try to refine the mappings in light of the latest quality estimates. */
16:  $\{v_{MH2}^r, \mu_{MH2}^r\} := \text{refine}(\{v_{MH2}^a, \mu_{MH2}^a\})$ 
17: /* Now, when we reevaluate the query  $q$ , we can use the refined mappings. */
18:  $r_4 := \text{evalIQ}(q, m_{MH2}, \{v_{MH2}^r, \mu_{MH2}^r\})$ 
19: /* Of course, selection and refinement steps can be interleaved as needed. */

```

Fig. 12.9 Bioinformatics use case: Improvement by gathering feedback on query results then using it for mapping selection and refinement.

12.6 Conclusion and Open Issues

In this work, we have built on our recent research on dataspace [6, 7, 33, 34, 35, 36, 50] and have aimed to present in some detail:

1. a view of dataspace management systems as second-generation data integration platforms in which pervasive automation is deployed to push down costs and user feedback is gathered and used to increase result quality;
2. a view of the life cycle of a dataspace that captures, albeit coarsely, the functionality of a growing body of literature (which we have most recently surveyed in [34]), is consistent with (I) and consisting of initialization, use, maintenance and improvement stages;
3. a conceptualization of the functional architecture of dataspace management systems that is consistent with (II) and explains more precisely than has been done so far in what ways they are related to classical database, data integration and model management systems;

4. a formalization of the functional model for dataspace management systems as a many-sorted algebra that is consistent with (3), and, while building on a history of advances by data integration and model management researchers, gives crisp contours to the notion of a dataspace, contours that had been hitherto only ambiguously and vaguely drawn;
5. examples of algebraic programs that illustrate how the functional model in (4) can capture and support the various stages of the data space life cycle in (2) as well as a more extended use case in bioinformatics that shows how important practical scenarios can be supported.

There is much still to accomplish in dataspace research, even though the literature, as shown here, has grown in breadth and depth and is very much thriving at the time of writing. Among the areas for further work in which the contribution we have made in this work may play a helpful role, we identify (without in any way aiming to be exhaustive):

1. refining and completing the formalization;
2. studying the mathematical and computational properties of the algebra;
3. proposing concrete algorithms for the algebraic operations;
4. implementing a dataspace management system that can be construed as an engine for the programs our algebra gives rise to;
5. considering rewriting strategies for heuristic optimization based on the mathematical properties of the algebra and cost-based optimization techniques based on the concrete algorithms available;
6. encompassing more forms of use;
7. studying the cost-benefit properties of more forms of feedback;
8. providing a principled treatment of how uncertainty can be quantified in relation to the evidence used and then propagated through operator applications.

Dataspaces are, in many respects, still but a promise, and it is too early to tell whether the belief that they could be effective and efficient in complementing classical database and data integration systems will be vindicated. The work we presented here aims to make the issues and challenges clearer and to constitute a step towards the exploration of this exciting hypothesis.

Acknowledgements. This work has been funded by the UK EPSRC under Grant EP/F031092/1. We are grateful for this support.

References

1. Alexe, B., Chiticariu, L., Miller, R.J., Tan, W.C.: Muse: Mapping Understanding and deSign by Example. In: ICDE, pp. 10–19. IEEE (2008)
2. Atzeni, P., Bellomarini, L., Bugiotti, F., Gianforme, G.: MISM: A Platform for Model-Independent Solutions to Model Management Problems. In: Spaccapietra, S., Delcambre, L. (eds.) Journal on Data Semantics XIV. LNCS, vol. 5880, pp. 133–161. Springer, Heidelberg (2009)

3. Atzeni, P., Cappellari, P., Torlone, R., Bernstein, P.A., Gianforme, G.: Model-Independent Schema Translation. *VLDB J.* 17(6), 1347–1370 (2008)
4. Aumueller, D., Do, H.H., Massmann, S., Rahm, E.: Schema and Ontology Matching with COMA++. In: Özcan, F. (ed.) *SIGMOD Conference*, pp. 906–908. ACM (2005)
5. Batini, C., Lenzerini, M., Navathe, S.B.: A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Comput. Surv.* 18(4), 323–364 (1986)
6. Belhajjame, K., Paton, N.W., Embury, S.M., Fernandes, A.A.A., Hedeler, C.: Feedback-based Annotation, Selection and Refinement of Schema Mappings for Dataspaces. In: *EDBT*, pp. 573–584 (2010)
7. Belhajjame, K., Paton, N.W., Fernandes, A.A.A., Hedeler, C., Embury, S.M.: User Feedback as a First Class Citizen in Information Integration Systems. In: *CIDR* (2011)
8. Benson, D.A., Karsch-Mizrachi, I., Lipman, D.J., Ostell, J., Wheeler, D.L.: *GenBank. Nucleic Acids Research* 31(1), 23–27 (2003); *Databases in biology: Genbank*
9. Bernstein, P.A., Halevy, A.Y., Pottinger, R.: A Vision of Management of Complex Models. *SIGMOD Record* 29(4), 55–63 (2000)
10. Bernstein, P.A., Melnik, S.: Model Management 2.0: Manipulating Richer Mappings. In: Chan, C.Y., Ooi, B.C., Zhou, A. (eds.) *SIGMOD Conference*, pp. 1–12. ACM (2007)
11. Blunski, L., Dittrich, J.-P., Girard, O.R., Karakashian, S.K., Salles, M.A.V.: A Dataspace Odyssey: The iMeMex Personal Dataspace Management System (Demo). In: *CIDR*, pp. 114–119 (2007)
12. Boyd, M., Kittivoravitkul, S., Lazanitis, C., McBrien, P., Rizopoulos, N.: AutoMed: A BAV Data Integration System for Heterogeneous Data Sources. In: Persson, A., Stirna, J. (eds.) *CAiSE 2004. LNCS*, vol. 3084, pp. 82–97. Springer, Heidelberg (2004)
13. Boyd, M., McBrien, P.: Comparing and Transforming Between Data Models Via an Intermediate Hypergraph Data Model. In: Spaccapietra, S. (ed.) *Journal on Data Semantics IV. LNCS*, vol. 3730, pp. 69–109. Springer, Heidelberg (2005)
14. Bult, C., Eppig, J., Kadin, J., Richardson, J., Blake, J., the members of the Mouse Genome Database Group: The Mouse Genome Database (MGD): Mouse Biology and Model Systems. *Nucleic Acids Research* 36(Database issue), D724–D728 (2008)
15. Cafarella, M.J., Halevy, A.Y., Khoussainova, N.: Data Integration for the Relational Web. *PVLDB* 2(1), 1090–1101 (2009)
16. Cao, H., Qi, Y., Candan, K.S., Sapino, M.L.: Feedback-driven Result Ranking and Query Refinement for Exploring Semi-structured Data Collections. In: *EDBT*, pp. 3–14 (2010)
17. Chiticariu, L., Hernández, M.A., Kolaitis, P.G., Popa, L.: Semi-Automatic Schema Integration in Clio. In: *VLDB*, pp. 1326–1329 (2007)
18. Das Sarma, A., Dong, X., Halevy, A.: Bootstrapping Pay-As-You-Go Data Integration Systems. In: *SIGMOD*, pp. 861–874 (2008)
19. Dittrich, J.-P., Vaz Salles, M.A.: iDM: A Unified and Versatile Data Model for Personal Dataspace Management. In: *VLDB*, pp. 367–378 (2006)
20. Do, H.-H., Rahm, E.: COMA: A System for Flexible Combination of Schema Matching Approaches. In: *VLDB*, pp. 610–621 (2002)
21. Doan, A., Halevy, A.Y.: Semantic Integration Research in the Database Community: A Brief Survey. *AI Magazine* 26(1), 83–94 (2005)
22. Doan, A., Ramakrishnan, R., Chen, F., DeRose, P., Lee, Y., McCann, R., Sayyadian, M., Shen, W.: Community Information Management. *IEEE Data Eng. Bull.* 29(1), 64–72 (2006)
23. Dong, X., Halevy, A.Y.: A Platform for Personal Information Management and Integration. In: *CIDR*, pp. 119–130 (2005)

24. Dong, X., Halevy, A.Y., Yu, C.: Data Integration with Uncertainty. In: VLDB, pp. 687–698 (2007)
25. Dong, X.L., Halevy, A.Y., Yu, C.: Data Integration with Uncertainty. VLDB J. 18(2), 469–500 (2009)
26. Fliccek, P., Aken, B.L., Ballester, B., et al.: Ensembl's 10th Year. Nucleic Acids Research 38(Database issue), D557–D562 (2010)
27. Franklin, M.J., Halevy, A.Y., Maier, D.: From Databases to Dataspaces: A New Abstraction for Information Management. SIGMOD Record 34(4), 27–33 (2005)
28. Haas, L.M., Lin, E.T., Roth, M.A.: Data Integration through Database Federation. IBM Systems Journal 41(4), 578–596 (2002)
29. Halevy, A.Y.: Answering Queries using Views: A Survey. The VLDB Journal 10(4), 270–294 (2001)
30. Halevy, A.Y., Franklin, M.J., Maier, D.: Principles of Dataspace Systems. In: Vansumeren, S. (ed.) PODS, pp. 1–9. ACM (2006)
31. Halevy, A.Y., Rajaraman, A., Ordille, J.J.: Data Integration: The Teenage Years. In: Dayal, U., Whang, K.Y., Lomet, D.B., Alonso, G., Lohman, G.M., Kersten, M.L., Cha, S.K., Kim, Y.K. (eds.) VLDB, pp. 9–16. ACM (2006)
32. Hedeler, C., Belhajjame, K., Fernandes, A.A.A., Embury, S.M., Paton, N.W.: Dimensions of Dataspaces. In: Sexton, A.P. (ed.) BNCOD 26. LNCS, vol. 5588, pp. 55–66. Springer, Heidelberg (2009)
33. Hedeler, C., Belhajjame, K., Mao, L., Paton, N.W., Fernandes, A.A.A., Guo, C., Embury, S.M.: Flexible Dataspace Management Through Model Management. In: EDBT/ICDT Workshops (2010)
34. Hedeler, C., Belhajjame, K., Paton, N.W., Campi, A., Fernandes, A.A.A., Embury, S.M.: Dataspaces. In: Ceri, S., Brambilla, M. (eds.) Search Computing. LNCS, vol. 5950, pp. 114–134. Springer, Heidelberg (2010)
35. Hedeler, C., Belhajjame, K., Paton, N.W., Fernandes, A.A.A., Embury, S.M., Mao, L., Guo, C.: Pay-As-You-Go Mapping Selection in Dataspaces. In: SIGMOD, pp. 1279–1282 (2011)
36. Hedeler, C., Paton, N.W.: Utilising the MISM Model Independent Schema Management Platform for Query Evaluation. In: Fernandes, A.A.A., Gray, A.J.G., Belhajjame, K. (eds.) BNCOD 2011. LNCS, vol. 7051, pp. 108–117. Springer, Heidelberg (2011)
37. Hernández, M.A., Ho, H., Popa, L., Fuxman, A., Miller, R.J., Fukuda, T., Papotti, P.: Creating Nested Mappings with Clío. In: ICDE, pp. 1487–1488 (2007)
38. Howe, B., Maier, D., Rayner, N., Rucker, J.: Quarrying Dataspaces: Schemaless Profiling of Unfamiliar Information Sources. In: ICDE Workshops, pp. 270–277 (2008)
39. Ives, Z.G., Green, T.J., Karvounarakis, G., Taylor, N.E., Tannen, V., Talukdar, P.P., Jacob, M., Pereira, F.: The ORCHESTRA Collaborative Data Sharing System. SIGMOD Record 37(3), 26–32 (2008)
40. Ives, Z.G., Knoblock, C.A., Minton, S., Jacob, M., Talukdar, P.P., Tuchinda, R., Ambite, J.L., Muslea, M., Gazen, C.: Interactive Data Integration through Smart Copy & Paste. In: CIDR (2009), www.crdldb.org
41. Jeffery, S.R., Franklin, M.J., Halevy, A.Y.: Pay-As-You-Go User Feedback for Dataspace Systems. In: SIGMOD, pp. 847–860. (2008)
42. Kanehisa, M., Goto, S., Furumichi, M., Tanabe, M., Hirakawa, M.: KEGG for Representation and Analysis of Molecular Networks Involving Diseases and Drugs. Nucleic Acids Research 38(Database issue), D355–D360 (2010)

43. Kensché, D., Quix, C., Li, X., Li, Y., Jarke, M.: Generic Schema Mappings for Composition and Query Answering. *Data Knowl. Eng* 68(7), 599–621 (2009)
44. Kim, W., Seo, J.: Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer* 24(12), 12–18 (1991)
45. Lenzerini, M.: Data Integration: A Theoretical Perspective. In: Popa, L. (ed.) *PODS*, pp. 233–246. ACM (2002)
46. Leser, U., Naumann, F.: (Almost) Hands-off Information Integration for the Life Sciences. In: *CIDR*, pp. 131–143 (2005)
47. Liu, J., Dong, X., Halevy, A.: Answering Structured Queries on Unstructured Data. In: *WebDB*, pp. 25–30 (2006)
48. Lorenzo, G.D., Hacid, H., Paik, H.Y., Benatallah, B.: Data Integration in Mashups. *SIGMOD Record* 38(1), 59–66 (2009)
49. Madhavan, J., Cohen, S., Dong, X.L., Halevy, A.Y., Jeffery, S.R., Ko, D., Yu, C.: Web-scale data integration: You can afford to pay as you go. In: *CIDR*, pp. 342–350 (2007)
50. Mao, L., Belhajjame, K., Paton, N.W., Fernandes, A.A.A.: Defining and Using Schematic Correspondences for Automatically Generating Schema Mappings. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) *CAiSE 2009*. LNCS, vol. 5565, pp. 79–93. Springer, Heidelberg (2009)
51. McCann, R., Shen, W., Doan, A.: Matching Schemas in Online Communities: A Web 2.0 Approach. In: *ICDE*, pp. 110–119 (2008)
52. McKusick, V.A.: Mendelian Inheritance in Man and Its Online Version, OMIM. *Am. J. Hum. Genet.* 80(4), 588–604 (2007), <http://www.ncbi.nlm.nih.gov/omim/>
53. Mecca, G., Papotti, P., Raunich, S., Buoncristiano, M.: Concise and Expressive Mappings with +Spicy. *PVLDB* 2(2), 1582–1585 (2009)
54. Melnik, S.: *Generic Model Management*. LNCS, vol. 2967. Springer, Heidelberg (2004)
55. Melnik, S., Bernstein, P.A., Halevy, A., Rahm, E.: A Semantics for Model Management Operators. Technical Report MSR-TR-2004-59, Microsoft Research (2004)
56. Melnik, S., Bernstein, P.A., Halevy, A., Rahm, E.: Supporting Executable Mappings in Model Management. In: *SIGMOD*, pp. 167–178 (2005)
57. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: A Programming Platform for Generic Model Management. In: *SIGMOD*, pp. 193–204 (2003)
58. Miller, R.J., Haas, L.M., Hernández, M.A.: Schema Mapping as Query Discovery. In: *VLDB*, pp. 77–88 (2000)
59. Miller, R.J., Hernández, M.A., Haas, L.M., Yan, L., Ho, C.T.H., Fagin, R., Popa, L.: The Clio Project: Managing Heterogeneity. *SIGMOD Record* 30(1), 78–83 (2001)
60. Parkinson, H., Sarkans, U., Kolesnikov, N., et al.: ArrayExpress Update - an Archive of Microarray and High-Throughput Sequencing-based Functional Genomics Experiments. *Nucleic Acids Research* (2010)
61. Poulouvasilis, A., McBrien, P.: A General Formal Framework for Schema Transformation. *Data Knowl. Eng.* 28(1), 47–71 (1998)
62. Rahm, E., Bernstein, P.A.: A Survey of Approaches to Automatic Schema Matching. *VLDB J.* 10(4), 334–350 (2001)
63. Sarma, A.D., Dong, X. L., Halevy, A. Y.: Data Modeling in Dataspace Support Platforms. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) *Mylopoulos Festschrift*. LNCS, vol. 5600, pp. 122–138. Springer, Heidelberg (2009)
64. Talukdar, P.P., Ives, Z.G., Pereira, F.: Automatically Incorporating New Sources in Keyword Search-based Data Integration. In: Elmagarmid, A.K., Agrawal, D. (eds.) *SIGMOD Conference*, pp. 387–398. ACM (2010)
65. Talukdar, P.P., Jacob, M., Mehmood, M.S., Crammer, K., Ives, Z.G., Pereira, F., Guha, S.: Learning to Create Data-Integrating Queries. *PVLDB* 1(1), 785–796 (2008)

66. The Gene Ontology Consortium: Gene Ontology: Tool for the Unification of Biology. *Nature Genetics* 25(1), 25–29 (2000); *Databases in Biology: Gene Ontology*
67. Vaz Salles, M.A., Dittrich, J.-P., Karakashian, S.K., Girard, O.R., Blunski, L.: iTrails: Pay-as-you-go Information Integration in Dataspaces. In: *VLDB*, pp. 663–674 (2007)
68. Vizcaíno, J.A., Côté, R., Reisinger, F., Foster, J.M., Mueller, M., Rameseder, J., Hermjakob, H., Martens, L.: A Guide to the Proteomics Identifications Database Proteomics Data Repository. *Proteomics* 9(18), 4276–4283 (2009)

Author Index

- Balke, Wolf-Tilo 15
Belhajjame, Khalid 305
Belussi, Alberto 83
Boucelma, Omar 271
Bressan, Stéphane 157
- Catania, Barbara 1, 83, 237
Colonna, François-Marie 271
- Embury, Suzanne M. 305
- Fernandes, Alvaro A.A. 305
- Gounaris, Anastasios 211
Guerrini, Giovanna 129, 237
Guo, Chenjuan 305
- Hedeler, Cornelia 305
Hwang, Seung-won 37
- Jain, Lakhmi 1
- Koutrika, Georgia 57
- Lofi, Christoph 15
- Manolopoulos, Yannis 211
Mao, Lu 305
Migliorini, Sara 83
- Nachouki, Gilles 271
- Ooi, Beng Chin 187
- Paton, Norman W. 305
Pitoura, Evaggelia 57
- Quafafou, Mohamed 271
- Stefanidis, Kostas 57
- Tan, Kian-Lee 187
Tok, Wee Hyong 157
Tsamoura, Efthymia 211
- Wu, Sai 187

Subject Index

- adaptive technique
 - for computation sharing 261
 - for distributed query optimization 231
 - for filters at remote data sources 260
 - for load balancing 261
 - for load management 220–230
 - for load shedding 260
 - for non-pipelined plan 257
 - for operator scheduling 149, 259
 - for pipelined plan 241–242, 258–259
 - for query parallelization 231
 - for query plan selection 255–259
 - for space partitioning 200–202
 - routing-based 256–257
- AdQP - Adaptive Query Processing *see* query processing, adaptive
- algorithm
 - branch and bound 97, 100–103, 107
 - partitioning-based 41
 - sorted-order 40
- ApQP - Approximate Query Processing
 - see* query processing, approximate
- BGLAV - Brigham Young University
 - Global-Local-as-View 277–280, 295–297
- bioinformatics *see* Genomics, 329–335
- cardinal relation 88, 118
- context
 - matching 69–70
 - specification 59–61
- CQP - Continuous Query Processing
 - see* query processing, continuous
- course of dimensionality 16
- data conflict 274–277
- data fusion 281–294
- data heterogeneity 89, 131, 274, 305
- data integration 3, 10, 272, 306
 - first-generation 272, 306
 - pay-as-you-go 10, 273, 306
 - second-generation 273, 306
- data reduction *see* summary, 116–118
- data scrambler 199–200
- dataspace 11, 273, 306
 - functional model 313–329
 - life cycle 307–309
 - management system 272, 310–312
 - operation 327–329
- data stream 8, 158, 189, 238, 243, 252
- data warehousing system 8, 187
- direction-based relation *see* cardinal relation
- distance-based relation 88, 115–116
- dominance 5, 18, 47, 91
 - k-dominance 23
 - spatial 105
 - weak 21–23, 51
- eddy 215–216, 257
 - distributed 218–219
- empty answer problem 5, 85, 243
- flushing policy 166, 168–169
- Flux operator 222–226

- GAV - Global as View 272
- Genomics 275, 294–300
- GIS - Geographic Information System 83
- HDS - Heterogeneous Data Sources 3, 10, 272, 305
- heuristic search 92, 113, 254
- histogram 251
- index-based data structure 84, 91, 163, 251
- Information Retrieval 130
- LAV - Local as View 272
- load management
 - adaptive *see* adaptive technique, for load management
 - cooperative 227–229
 - inter-operator 226–229
 - intra-operator 221–226
 - non-cooperative 229
- load shedding 253
- logical data independence 1
- many answer problem 5, 85
- MAPE architecture 214, 255
- MapReduce 207–208
- match-based similarity 144–145
- mediator 10, 272, 306, 308
- model management 307
 - operation 325–327
 - system 312–313
- non-pipelined plan 216, 242
- online aggregation 8, 189
 - for multi-relation queries 194–197
 - for multiple queries 197–199
 - in distributed environments 203–204
 - in MapReduce 207–208
- Pareto semantics *see* dominance
 - relaxed *see* dominance, weak
- performance evaluation 54, 172–174, 300
- pipelined plan 216, 242
- PQ - Preference-based Queries *see* query, preference-based
- PQP - Preference-based Query
 - Processing *see* query processing, preference-based
- preference
 - composition of 64–66
 - conditional network of 63
 - conflicting 64
 - contextual 59
 - elicitation 31–32
 - hybrid model 77
 - implicit 66
 - learning 76–77
 - of a user group 78
 - on attribute 63
 - on tuple 62
 - probabilistic model 78
 - qualitative 5, 61, 76
 - quantitative 5, 61, 76
 - relevance 70–71
 - representation 59–68
 - selection 68–73
 - specification 61–64
- profile tree 71–72
- progressive technique
 - for approximate join 170–180
 - for join 158–160, 194–197
 - for relational join 161–162
 - for similarity join 164
 - for spatial join 163–164
 - for XML structural join 164
 - join framework 164–170
- QoD - Quality of Data 3, 239, 244
- QoS - Quality of Service 3, 239, 249, 254
- QR - Query Relaxation 2, 6, 89–92, 138–142, 245
- query
 - continuous *see* progressive technique, 8, 213
 - multiway spatial join 113–115
 - nearest neighbor 88
 - preference-based 3, 58, 91, 246–247
 - similarity join 248
 - skyline 5, 16, 47, 91, 243–244, 246–247
 - spatial join 88
 - spatial selection 88
 - spatial skyline 104–112, 123–124
 - spatial top-k 93–103, 121–123
 - top-*k* 5, 26–32, 42–43, 91, 137, 246
 - twig 132–137

- query optimization 230–231
 - cost-based 45–47, 53–54, 213
 - dynamic 214, 254
 - static 211, 214
- query personalization 5, 68–75
- query processing
 - adaptive 3, 9, 131, 148, 149, 152, 196, 200, 212, 254–261
 - approximate 2, 6, 92–93, 112–118, 150, 170–180, 188, 248–254
 - centralized 9, 215–216
 - continuous 3, 8, 158, 189, 243, 246, 253, 256, 260
 - distributed 9, 203–207, 212
 - of heterogeneous data sources 278–280, 287–294
 - personalized 73–75
 - preference-based 3, 5, 39, 58, 96–103, 106–112, 148–151
- query rewriting 89–90, 213, 245
- ranking function 28, 42, 61, 91, 94–96, 142–148, 246
- routing policy 215–216, 218–220
- sampling 8, 175, 177–180, 190, 193–194, 251, 325
 - distributed 204–205
- semantic mapping 278–279, 283–287, 308, 319–321, 326–328
- semantic reconciliation *see* data fusion
- sketch 252
- skyline 18
 - approximately dominating representative 23–24
 - frequency 27
 - k-most representative points 29
 - level order 21
 - query *see* query, skyline
 - sampling 24–25
 - skycube 26–28, 55
 - SKYRANK 28–29
 - spatial *see* query, spatial skyline
 - summarization 23–25
 - top-k frequent 27
 - trade-off 32–33
- source discovery 324–325
- spatial data 6, 85–86
- spatial relation 86–88
- statistical model 192–193
- structural approximation 140–142
- structure filter 146–148
- summary 250–253
- synopsis 252
- td-idf scoring 145–146
- top-k
 - frequent skyline *see* skyline, top-k frequent
 - personalized retrieval 30–31, 68
 - query *see* query, top-k
- topological relation 86–88, 118
- tree-edit distance 143–144
- user feedback 25, 31–32, 321, 327–328
- user profile 59
- vocabulary approximation 138–139
- wavelet 251
- Web 273
- XML 6, 130, 133, 164, 245, 246, 248, 259, 277

Editors



Barbara Catania is Associate Professor at the Department of Computer and Information Sciences of the University of Genova, Italy. From 1999 until April 2004, she has been an assistant professor at the same department. She received the MS degree in Information Sciences from the University of Genoa, Italy, in 1993. In 1998, she received a Ph.D. in Computer Science from the University of Milan, Italy. She has been Visiting Researcher at the European Computer-Industry Research Center of Bull, ICL, and Siemens in Munich, Germany, and at the National University of Singapore. Her research activities are related to various aspects of data management, including: pattern management, geo-spatial databases, query processing and indexing techniques for advanced models and applications, deductive and constraint databases, access control. She is a member of the ACM and the IEEE.



Professor Lakhmi C. Jain is a Director/Founder of the Knowledge-Based Intelligent Engineering Systems (KES) Centre, located in the University of South Australia. He is a fellow of the Institution of Engineers Australia.

His interests focus on the artificial intelligence paradigms and their applications in complex systems, art-science fusion, virtual systems, e-education, e-healthcare, unmanned air vehicles and intelligent agents.