Processing Rank-Aware Queries in Schema-Based P2P Systems

Dissertation
zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt der
Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von
Dipl.-Inf. Katja Hose

Tag der Einreichung: 27. Oktober 2008
Tag der wissenschaftlichen Aussprache: 17. April 2009

Gutachter

1. Kai-Uwe Sattler
2. Alon Halevy
3. Felix Naumann

urn:nbn:de:gbv:ilm1-2009000084

Processing Rank-Aware Queries in Schema-Based P2P Systems

Katja Hose

A dissertation submitted
in partial fulfillment of the requirements
for the degree of

Doktor-Ingenieur (Dr.-Ing.)

Faculty of Computer Science and Automation
Technische Universität Ilmenau

October 2008

Reading Committee

1. Kai-Uwe Sattler
2. Alon Halevy
3. Felix Naumann

Abstract

In recent years, there has been considerable research with respect to query processing in data integration and P2P systems. Conventional data integration systems consist of multiple sources with possibly different schemas, adhere to a hierarchical structure, and have a central component (mediator) that manages a global schema. Queries are formulated against this global schema and the mediator processes them by retrieving relevant data from the sources transparently to the user. Arising from these systems, eventually Peer Data Management Systems (PDMSs), or schema-based P2P systems respectively, have attracted attention. Peers participating in a PDMS can act both as a mediator and as a data source, are autonomous, and might leave or join the network at will. Due to these reasons peers often hold incomplete or erroneous data sets and mappings. The possibly huge amount of data available in such a network often results in large query result sets that are hard to manage. Due to these reasons, retrieving the complete result set is in most cases difficult or even impossible. Applying rank-aware query operators such as top-N and skyline, possibly in conjunction with approximation techniques, is a remedy to these problems as these operators select only those result records that are most relevant to the user. Being aware that in most cases only a small fraction of the complete result set is actually output to the user, retrieving the complete set before evaluating such operators is obviously inefficient.

Therefore, the questions we want to answer in this dissertation are how to compute such queries in PDMSs and how to do that efficiently. We propose strategies for efficient query processing in PDMSs that exploit the characteristics of rank-aware queries and optionally apply approximation techniques. A peer's relevance is determined on two levels: on schema-level and on data-level. According to its relevance a peer is either considered for query processing or not. Because of heterogeneity queries need to be rewritten, enabling cooperation between peers that use different schemas. As existing query rewriting techniques mostly consider conjunctive queries only, we present an extension that allows for rewriting queries involving rank-aware query operators. As PDMSs are dynamic systems and peers might update their local data, this dissertation addresses not only the problem of considering such structures within a query processing strategy but also the problem of keeping them up-to-date. Finally, we provide a system-level evaluation by presenting SmurfPDMS (SiMUlating enviRonment For Peer Data Management Systems) – a system created in the context of this dissertation implementing all presented techniques.

Zusammenfassung

Effiziente Anfragebearbeitung in Datenintegrationssystemen sowie in P2P-Systemen ist bereits seit einigen Jahren ein Aspekt aktueller Forschung. Konventionelle Datenintegrationssysteme bestehen aus mehreren Datenquellen mit ggf. unterschiedlichen Schemata, sind hierarchisch aufgebaut und besitzen eine zentrale Komponente: den Mediator, der ein globales Schema verwaltet. Anfragen an das System werden auf diesem globalen Schema formuliert und vom Mediator bearbeitet, indem relevante Daten von den Datenquellen transparent für den Benutzer angefragt werden. Aufbauend auf diesen Systemen entstanden schließlich Peer-Daten-Management-Systeme (PDMSs) bzw. schemabasierte P2P-Systeme. An einem PDMS teilnehmende Knoten (Peers) können einerseits als Mediatoren agieren andererseits jedoch ebenso als Datenquellen. Darüber hinaus sind diese Peers autonom und können das Netzwerk jederzeit verlassen bzw. betreten. Die potentiell riesige Datenmenge, die in einem derartigen Netzwerk verfügbar ist, führt zudem in der Regel zu sehr großen Anfrageergebnissen, die nur schwer zu bewältigen sind. Daher ist das Bestimmen einer vollständigen Ergebnismenge in vielen Fällen äußerst aufwändig oder sogar unmöglich. In diesen Fällen bietet sich die Anwendung von Top-N- und Skyline-Operatoren, ggf. in Verbindung mit Approximationstechniken, an, da diese Operatoren lediglich diejenigen Datensätze als Ergebnis ausgeben, die aufgrund nutzerdefinierter Ranking-Funktionen am relevantesten für den Benutzer sind. Da durch die Anwendung dieser Operatoren zumeist nur ein kleiner Teil des Ergebnisses tatsächlich dem Benutzer ausgegeben wird, muss nicht zwangsläufig die vollständige Ergebnismenge berechnet werden sondern nur der Teil, der tatsächlich relevant für das Endergebnis ist.

Die Frage ist nun, wie man derartige Anfragen durch die Ausnutzung dieser Erkenntnis effizient in PDMSs bearbeiten kann. Die Beantwortung dieser Frage ist das Hauptanliegen dieser Dissertation. Zur Lösung dieser Problemstellung stellen wir effiziente Anfragebearbeitungsstrategien in PDMSs vor, die die charakteristischen Eigenschaften ranking-basierter Operatoren sowie Approximationstechniken ausnutzen. Peers werden dabei sowohl auf Schema- als auch auf Datenebene hinsichtlich der Relevanz ihrer Daten geprüft und dementsprechend in die Anfragebearbeitung einbezogen oder ausgeschlossen. Durch die Heterogenität der Peers werden Techniken zum Umschreiben einer Anfrage von einem Schema in ein anderes nötig. Da existierende Techniken zum Umschreiben von Anfragen zumeist nur konjunktive Anfragen betrachten, stellen wir eine Erweiterung dieser Techniken vor, die Anfragen mit ranking-basierten Anfrageoperatoren berücksichtigt. Da PDMSs dynamische Systeme sind und teilnehmende Peers jederzeit ihre Daten ändern können, betrachten wir in dieser Dissertation nicht nur wie Routing-Indexe verwendet werden, um die Relevanz eines Peers auf Datenebene zu bestimmen, sondern auch wie sie gepflegt werden können. Schließlich stellen wir SmurfPDMS (SiMUlating enviRonment For Peer Data Management Systems) vor, ein System, welches im Rahmen dieser Dissertation entwickelt wurde und alle vorgestellten Techniken implementiert.

Acknowledgements

There are a couple of people I want to thank. First, I would like to thank Kai-Uwe Sattler not only for his guidance and for supervising the project but also for offering me a PhD position after I graduated – without the offer I might never have started a career in database research. Second, I would like to thank Alon Halevy and Felix Naumann for many discussions and for the feedback they have given me. Another person who paved the way for me to become a computer scientist is Rainer Werner. It was he who helped me to get along with my first PC – especially after I have carefreely tried out some DOS commands without knowing what they actually did – the format command in particular had a surprising and long-lasting effect. I am also grateful to my parents and my sister for encouraging me to keep going no matter how much effort it cost to reach my goal. I would also like to thank Marcel Karnstedt and all other fellow PhD students at the department who joined me on the way to receive a PhD for those many discussions and support in all aspects. I would like to particularly mention Anja Fischer for reading several drafts of this dissertation. I also appreciate the help of the students whose diploma theses I supervised and who contributed to the development of SmurfPDMS: Christian Lemke, Jana Quasebarth, Daniel Zinn, and Andreas Job.

Contents

Abstract . . . v
Zusammenfassung . . . vii
Acknowledgements . . . ix
List of Figures . . . xv
List of Tables . . . xix

1 Introduction **1**
1.1 PDMS System Definition . . . 3
1.2 Distributed Query Processing in PDMSs . . . 6
1.3 Outline of Dissertation . . . 9

2 Background **11**
2.1 Rank-Aware Query Operators . . . 11
2.1.1 Top-N Queries . . . 11
2.1.2 Skyline Queries . . . 14
2.1.3 Applicability to PDMSs . . . 22
2.2 Approximation . . . 23
2.2.1 Pruning-Based Approximation . . . 24
2.2.2 Representation-Based Approximation . . . 25
2.2.3 Applicability to PDMSs . . . 26
2.3 Query Routing and Routing Indexes . . . 27
2.3.1 Routing Indexes . . . 28
2.3.2 Histograms and Bloom Filters as Routing Indexes . . . 30
2.3.3 Bit Vectors as Routing Indexes . . . 31
2.3.4 Routing Indexes Based on Indexing Clusters . . . 32
2.3.5 Applicability to PDMSs . . . 33
2.4 Histograms and Maintenance . . . 34
2.4.1 Maintenance . . . 35
2.4.2 Applicability to PDMSs . . . 37
2.5 Distributed Query Processing Strategies . . . 37
2.5.1 Data Shipping and Query Shipping . . . 38
2.5.2 Mutant Query Plans . . . 39
2.5.3 Incremental Message Shipping . . . 39
2.5.4 Applicability to PDMSs . . . 41
2.6 Query Rewriting . . . 41
2.6.1 Conjunctive Queries . . . 44
2.6.2 Query Containment and Containment Mapping . . . 46

2.6.3 Query Rewriting Algorithms 48
2.6.4 Applicability to PDMSs 50
2.7 PDMS Implementations 51
2.7.1 Piazza 52
2.7.2 Hyperion 53
2.7.3 System P 53
2.7.4 HePToX 54
2.7.5 coDB 54
2.7.6 Orchestra 55
2.8 Conclusion 55

3 Model Definition 59
3.1 Network Model 60
3.2 Mapping Definition Language 60
3.3 Query Formulation 64
3.4 Summary 68

4 Rewriting Rank-Aware Queries for XML Data 71
4.1 Subgoal Trees 72
4.1.1 Creating Subgoal Trees for Views 72
4.1.2 Creating Subgoal Trees for Queries 77
4.2 Query Rewriting Using Subgoal Trees 79
4.2.1 Receiving a Query Plan 80
4.2.2 Preprocessing 81
4.2.3 Creating Buckets 84
4.2.4 Sorting View Subgoals into Buckets 85
4.2.5 Creating Combinations of Buckets 86
4.2.6 Creating Query Snippets 88
4.2.7 Creating Initial Rewritings 89
4.2.8 Optimizing Remote Queries and Rewritings 91
4.2.9 Assembling the Rewritten Query Plan 94
4.3 Cycle Detection 95
4.4 Evaluation 99
4.4.1 Rewriting vs. Schema Indexing 99
4.4.2 The Influence of the Choice of Neighbors 102
4.4.3 Benefits of Considering Rank-Aware Query Operators for Rewriting 104
4.4.4 Costs for Rewriting 105
4.5 Conclusion 107

5 Query Routing 109
5.1 Distributed Data Summaries 110
5.2 QTree 112
5.2.1 Definition 113
5.2.2 Construction 113
5.2.3 Lookups 119
5.2.4 Deletions 120

5.2.5 Penalty Functions 126
5.2.6 Evaluation 130
5.2.7 Extension to Support String Attributes 136
5.2.8 Summary 138
5.3 Maintenance 138
5.3.1 Update Representation and Rewriting Updates as well as Summaries 139
5.3.2 Constructing Distributed Data Summaries from Scratch 141
5.3.3 Classification of Maintenance Strategies 145
5.3.4 Update-Driven Strategies 147
5.3.5 Query-Driven Strategies 154
5.3.6 Comparison 161
5.3.7 Evaluation 163
5.4 Conclusion 169

6 Query Processing **171**
6.1 Processing Top-N Queries Using DDSs 173
6.1.1 Top-N Queries on Regions 173
6.1.2 Distributed Processing of Top-N Queries 175
6.2 Skyline Query Processing 177
6.2.1 Skylines on Regions 177
6.2.2 Distributed Processing of Skyline Queries 179
6.3 Constraints 181
6.4 Relaxation 182
6.4.1 Top-N 182
6.4.2 Skylines 187
6.5 Evaluation 190
6.5.1 Top-N Queries 191
6.5.2 Skyline Queries 195
6.6 Conclusion 196

7 SmurfPDMS **199**
7.1 System Architecture 200
7.1.1 Network Layer 201
7.1.2 Simulation Layer 201
7.1.3 GUI Layer 203
7.2 Simulation Core 205
7.3 Simulation Workflow 207
7.4 Important Steps of Simulating a PDMS 209
7.4.1 Detecting SmurfPDMS Instances 210
7.4.2 Initializing a Simulation 210
7.4.3 Running a Simulation 214
7.4.4 Evaluating Statistics and Creating Diagrams 218
7.5 Evaluation 219
7.6 Conclusion 220

8 Conclusion and Future Work **223**

List of Figures

1.1 Mediator System . . . 2
1.2 Peer Data Management System . . . 2
1.3 Example for Schema Heterogeneity . . . 4
1.4 PDMS as Data Integration System . . . 6
1.5 Distributed Query Processing . . . 7
1.6 Query Processing in a PDMS . . . 8

2.1 Top-N Query example . . . 12
2.2 Skyline Query Example . . . 14
2.3 Classification of Algorithms for Skyline Computation . . . 15
2.4 D&C Dominating Regions . . . 16
2.5 Constrained Skyline Query Example . . . 18
2.6 Classification of Approximation Techniques for Top-N and Skyline Queries 24
2.7 Classification of Routing in Unstructured P2P Systems . . . 27
2.8 Routing Indexes . . . 29
2.9 Query Shipping vs. Data Shipping . . . 38
2.10 Mutant Query Plans . . . 39
2.11 Incremental Message Shipping - Query Propagation . . . 40
2.12 Incremental Messsage Shipping - Answer Propagation . . . 40
2.13 GAV Example . . . 43
2.14 LAV Example . . . 44
2.15 Conjunctive Queries on Relations and XML Data . . . 46

3.1 Query Processing in PDMSs - Query Formulation/Parsing and Mapping Definition . . . 59
3.2 Example Network . . . 61
3.3 Correspondences between the Schemas of Two Peers . . . 62
3.4 Mappings between Two Peers . . . 62
3.5 POP Tree Representation . . . 68

4.1 Query Processing in PDMSs – Query Rewriting . . . 72
4.2 Creating Subgoal Trees for View Definitions . . . 74
4.3 Subgoal Trees for the Example Skyline Query . . . 77
4.4 Subgoal Trees for the Example Join Query . . . 78
4.5 Preprocessing of a Query with Union POPs . . . 81
4.6 Preprocessing of a Skyline Query . . . 82
4.7 Preprocessing of a Skyline Query with Join . . . 83

4.8 Preprocessing of a Skyline Query with Join – Alternative 84
4.9 Sorting View Subgoals into Buckets 87
4.10 Creating Query Snippets from Tagged Subgoal Trees 90
4.11 Generating Rewritings for a Skyline Query 90
4.12 Generating Rewritings for a Join Query 91
4.13 Optimizing Rewritings 92
4.14 Combining Remote Queries of a Rewriting 93
4.15 Removing a Join in a Rewriting 94
4.16 Reordering Joins in a Query 94
4.17 Combining Joins in a Query 95
4.18 Rewritten Query Plans as Results of the Rewriting Process 96
4.19 Cycle Detection 97
4.20 Example Network with Mappings 100
4.21 Example Network without Mappings 101
4.22 Example Network with 20 Peers 103
4.23 Benefits of Considering Rank-Aware Query Operators for Rewriting . . . 105

5.1 Query Processing in PDMSs – Distributed Data Summaries 110
5.2 Structure of a Routing Index 111
5.3 Equi-Width Histogram as Base Structure 112
5.4 Deletions in QTrees with Overlapping Buckets 121
5.5 Deletion of Buckets - Removing a Sibling Node 123
5.6 Deletion of Buckets - Removing the Parent Node 123
5.7 QTree – Problems when Deleting Records 124
5.8 Random Test Data 126
5.9 Random Test Data in a QTree Using $P_{MaxBound}$ as Penalty Function . . . 126
5.10 Random Test Data in a QTree Using P_{SumDim} as Penalty Function . . . 127
5.11 Random Test Data in a QTree Using P_{AvgDim} as Penalty Function 127
5.12 Random Test Data in a QTree Using $P_{Diameter}$ as Penalty Function . . . 128
5.13 Random Test Data in a QTree Using P_{Volume} as Penalty Function 128
5.14 Comparison of Penalty Functions – Random Data 129
5.15 Cluster Test Data 130
5.16 Comparison of Penalty Functions – Clustered Data 130
5.17 Influence of f_{max} and b_{max} on Random Data 131
5.18 Influences of Data Distribution and Dimensionality 133
5.19 Influence of the Order of Insertion on the Approximation Error 134
5.20 Influence of the Deletion Strategy – Deleting Single Records from the Random Data Set 134
5.21 Influence of the Deletion Strategy – Deleting Single Records from the Clustered Data Set 135
5.22 QTree after Deletion of 4800 of 5000 Records – Random Data 135
5.23 Rewriting Updates in the Presence of LAV Mappings 140
5.24 DDS Construction with P_1 as Initiator 143
5.25 DDS Construction in the Presence of Unidirectional Mappings 144
5.26 DDS Construction – 2-Phase-Flooding 145
5.27 Example for Update Propagation Using STPS 149

5.28 Example for Update Propagation Using ATPS 151
5.29 Example for Updating Routing Indexes Using Query Feedback 158
5.30 Exploiting Cached Knowledge about Queried Regions 159
5.31 Evaluation Results for Update-Driven Strategies (IPS, STPS, and ATPS) 164
5.32 False Routing Decisions for Update-Driven Strategies and a Propagation Rule Based on $\mathcal{M}_{NonProp}$ 165
5.33 Evaluation Results for QFS 167
5.34 Evaluation Results for QES 168

6.1 Query Processing in PDMSs – Query Optimization 172
6.2 Top-N Query Evaluation on Regions 175
6.3 Skyline on Regions 179
6.4 Relaxed Top-N Query Example 183
6.5 Finding Representatives for Regions 186
6.6 Relaxed Skyline Query Example 188
6.7 Test Data 191
6.8 Distributed Processing of Relaxed Top-N Queries 192
6.9 Cost Benefit Analysis for Top-N Query Processing 193
6.10 Distributed Processing of Relaxed Constrained Top-N Queries 194
6.11 Distributed Processing of Relaxed Skylines 196
6.12 Distributed Processing of Relaxed Constrained Skylines 197
6.13 Cost Benefit Analysis for Skyline Query Processing 198

7.1 SmurfPDMS System Architecture 201
7.2 SmurfPDMS – Class Diagram: Associations between Simulation Layer and Network Layer 202
7.3 SmurfPDMS GUI – Main Window 203
7.4 SmurfPDMS GUI – Simulation Windows 204
7.5 SmurfPDMS GUI – Statistics Window 205
7.6 SmurfPDMS – Simulation Core Architecture 206
7.7 SmurfPDMS – General Steps of Running a Simulation 208
7.8 SmurfPDMS – Important Classes of the Simulation Layer Instantiated During Runtime 209
7.9 SmurfPDMS – Topology and Data Generation 212
7.10 SmurfPDMS – Query Processing Parameters 213
7.11 SmurfPDMS – Communication Connections between SmurfPDMS Instances 214
7.12 SmurfPDMS – Data Updates Parameters 215
7.13 SmurfPDMS – Interactions between Classes for Synchronization 217
7.14 SmurfPDMS – Evaluation Environment 219
7.15 SmurfPDMS – Diagram Generation 220
7.16 SmurfPDMS – Performance of Distributed Simulations 221

List of Tables

2.1 Characteristics of PDMSs and their Spectrums of Values 51

3.1 Terminology 63
3.2 Algebra Operators 65

4.1 Evaluation Query Mix 100
4.2 Results for Query Shipping 101
4.3 Results for Incremental Message Shipping 102
4.4 Comparison of Query Processing in Data Integration Systems and PDMSs 104

5.1 Advantages and Disadvantages of the Strategies Discussed in this Section 161

Chapter 1

Introduction

In many application scenarios data is distributed among multiple sources. In order to access the data of all these sources, multiple queries have to be formulated. This is a very inconvenient solution, especially if the sources use different schemas to manage their data (schema heterogeneity). Data integration systems have been developed to make querying the sources more convenient. Their objective is to integrate the data in a way that makes querying it transparent to the user, i.e., to formulate only one query and have the system query the data sources automatically. A solution to query the sources in spite of schema heterogeneity in an automatic fashion is to formulate the original query on a uniform query interface using a mediated schema (global schema) and automatically rewrite the query into the schemas of the sources. For this purpose, the system needs to know the correspondences between schema elements of the source schemas and of the global schema (provided by mappings). An example for such a data integration system is a mediator system, which uses a mediator to provide the uniform query interface and to rewrite the queries formulated on the global schema. Figure 1.1 sketches the structure of such a system with each source being connected to the mediator. However, this approach has a bottleneck: the mediator. If it crashes, an application can no longer query remote data sources. Although algorithms might be applied to appoint another mediator as a backup solution (replication of the mediator), extensibility of such a system is still limited. Participants issuing queries still strongly depend on the mediator and have no noteworthy degree of autonomy.

In contrast, data sources (i.e., peers) in P2P systems have the utmost degree of autonomy. In such loosely-coupled systems peers can leave or join the network at any time. Peers are equal in terms of functionality and responsibilities, i.e., each peer participating in the network might issue queries and there is no peer that performs special tasks in comparison to the others. However, there are two main classes of P2P systems that we have to distinguish: structured and unstructured P2P systems. For the latter class the sovereignty over the data remains at the peer providing it, i.e., the data that a peer decides to "share" with the network remains at the peer and is neither relocated nor replicated at another peer. In contrast, in structured P2P systems the shared data is redistributed among participating peers according to a commonly known rule, e.g., a hash function. Another difference between these classes is that unstructured P2P systems do not require any specific network topology, i.e., a peer is free to choose its neighbors, whereas a structured P2P system is based on a fixed topology such as a hypercube [168],

a ring [180], or a tree [101]. Thus, in unstructured P2P systems peers are free to build semantic clusters of peers with similar schemas [49, 112, 128, 138].

The consequence of strict peer autonomy in unstructured P2P systems is that there is no global knowledge, i.e., there is no central instance that might help a peer to find relevant data for a particular query – whereas in structured systems the rule according to which the data has been redistributed in the first place can be used for this purpose. In unstructured systems peers have to make routing decisions independently from all the other peers. The question is how to route queries efficiently to only those peers that provide relevant data if there is no global knowledge or central instance that could help identify them. One means to overcome this disadvantage is the application of routing indexes [41], i.e., each peer maintains local summaries of the data that can be retrieved via its neighbors.

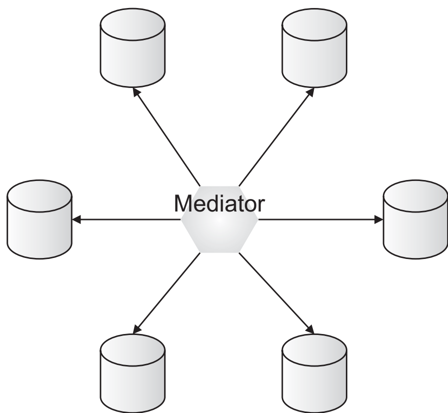


Figure 1.1: Mediator System

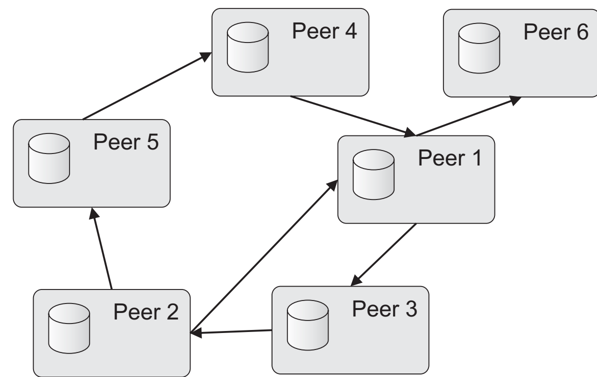


Figure 1.2: Peer Data Management System

By combining the two approaches (unstructured P2P systems and data integration) we obtain Peer Data Management Systems (PDMSs) – Figure 1.2 sketches an example. PDMSs can, for example, be applied in ad-hoc data integration scenarios such as disaster management [85] or as decentralized data structures for mediation between ontologies in the Semantic Web [5]. Another application of PDMSs proposed by the literature is to enable information exchange between enterprises or research institutes [95, 170]. With unstructured P2P systems as basis, PDMSs inherit characteristics such as a high degree of peer autonomy and the absence of both global knowledge and a single point of failure. In contrast to mediator systems, peers are considered equal so that each peer can issue queries as well as participate in answering them. Another difference in comparison to data integration systems is that paths from one peer to another are likely to involve more than two peers. Thus, queries may be transformed multiple times (chaining) before they reach distant peers holding relevant data.

An important aspect of query processing in PDMSs is minimizing execution costs. In contrast to centralized systems the most important cost factor in PDMSs is not local execution costs but network traffic and the number of involved peers. Nevertheless, a popular strategy for query processing in PDMSs (but also for unstructured P2P and data integration systems) is to forward the query to all the peers in the network (flooding) whose schemas indicate their relevance. However, in large-scale settings flooding the

network in order to answer a single query is extremely expensive or sometimes even impossible.

Thus, the goal of a query processing strategy in a PDMS is to prune peers from consideration (e.g., by applying routing indexes) that cannot contribute to the result. Whenever a peer is pruned from the set of peers that a query is forwarded to (i) local execution costs at the pruned peer are reduced to zero because it does not participate in answering the query, (ii) network traffic is reduced as no data is sent to or received from the pruned peer, and (iii) local execution costs at the initiator are reduced because there are less answer messages and thus less data to be processed locally in order to compute the final result. Hence, whenever a peer is pruned from the set of peers that the query is forwarded to, execution costs are reduced.

In most cases users are interested only in a quick overview of the data available in the network. Such an overview can be provided by applying rank-aware query operators such as top-N and skyline [19], which are already applied in centralized systems and P2P systems. As these operators reduce the result set, they also reduce the number of relevant peers and thus query execution costs. In consideration of PDMS characteristics such as the system's dynamic behavior and incomplete or erroneous mappings and data, it is already difficult to guarantee complete query answers. Thus, introducing approximation techniques that sacrifice result completeness/correctness in a controlled fashion up to a certain user-defined extent, is another possibility to reduce execution costs without much increasing the loss of completeness we have to deal with even without applying approximation.

What distinguishes PDMSs from other P2P systems the most is the fact that in a PDMS each peer is allowed to have its own data schema – a characteristic inherited from data integration systems. We need mappings and query rewriting algorithms to overcome data heterogeneity and translate a query formulated in one schema into another. If both are the same, there is no extra effort. If not – and this is assumed to be the normal case – the query has to be rewritten into the other schema exploiting the knowledge of schema correspondences given in the form of mappings in conjunction with appropriate query rewriting algorithms.

Operating PDMSs entails finding solutions to the problems arising from their main characteristics discussed above. Since the notion of PDMSs came up only a few years ago, research concerning PDMSs is still more or less in its infancy being an important issue of ongoing research. Although the quality of mappings plays a decisive role for a PDMS, the main focus of this dissertation is not mapping definition or network construction but the practical use of PDMSs, i.e., efficient techniques for query processing. Before motivating the challenges and the contributions of this dissertation, Sections 1.1 and 1.2 give a brief introduction to PDMSs and query processing in such systems.

1.1 PDMS System Definition

The basic prerequisite to building a network of largely autonomous peers is a peer's willingness to cooperate with others. This means at least some of the peer's data and resources have to be accessible to other peers connected via communication links. This is only a negligible impairment of peer autonomy that has to be accepted in order to build

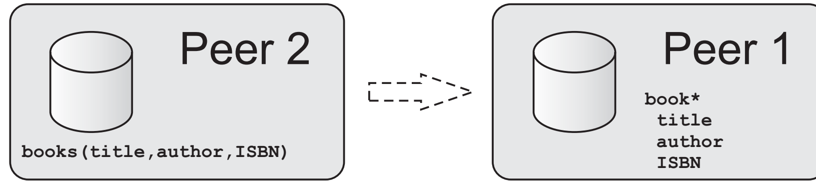


Figure 1.3: Different Schemas Representing the Same Kind of Data: Relations (left), XML (right)

a network of cooperating peers. Since PDMSs also inherit some characteristics from data integration systems (federated database systems [40]), let us review the degrees of peer autonomy that have been identified for these systems [166, 177]:

- **Design Autonomy:** The local schema can be designed by each peer independently from others.
- **Execution Autonomy:** Local data can be changed (deleted, added, updated) without interference from external operations. External queries (operations) are treated the same way as local queries (operations).
- **Communication Autonomy:** A peer with communication autonomy is able to decide when and how it responds to a request from another peer.
- **Association Autonomy:** Each peer has the ability to decide whether and to what extent to share its functionality (i.e., the operations it supports) and resources (i.e., the data it manages) with others. This includes the ability to associate or disassociate itself from the network and the ability to participate in one or more networks.
- **Hardware and Software Autonomy:** Each peer may choose its hardware and software in accordance to its local needs.

PDMSs aim at conserving as much autonomy as possible while allowing participants to share data and functionality. Being a consequence of design autonomy one of the main problems PDMSs have to deal with is heterogeneity. Whatever data model we use, there are always various possibilities to model the same semantic issue even though using the same model. Moreover, schemas may differ in their data models (relational, object-oriented, object-relational, XML). Take Figure 1.3 for an example. The left hand side shows a peer storing data in relations. The right hand side shows a peer storing the same kind of data in XML format - nesting is represented by indentation and "*" means the element may occur multiple times or never ($[0, *]$).

To reconcile all these different models in just one system with the aim of supporting data exchange, it is useful to appoint one model for peer interaction. Each peer using a different local data model can still participate by using wrappers so that in the end all the peers provide *peer schemas* in the same model describing their local data. In case a peer does not want to make all its local data available to other peers, its peer schema might represent only a portion of its local data. For simplicity, we will make no

distinction between a peer's peer schema and the schema the data is actually stored in locally – assuming both are the same.

In principle, we could still define a common global schema for all peers while allowing them to keep their local ones. However, using this kind of global schema for peer interaction has several shortcomings. It means to use global knowledge because all the peers would have to know and to use the same global schema. However, we consider dynamic systems whose peers might leave or join the network at any time. The global schema would ideally have to integrate all data that peers (including future peers) might want to share. If not, it would have to be adapted whenever a new peer with semantically unique data joins the network. The only option to do this efficiently is again a central component representing the bottleneck of the system.

Alternatively, peers can make their local data available by defining several binary mappings to other peers (neighbors) participating in the network. This is the paradigm we use in PDMSs so that the usage of global knowledge is avoided. In general, when using binary mappings between peers, each peer only knows correspondences to the schemas of its direct neighbors. By chained rewriting data from peers that are located in a distance of several hops can also be accessed. The resulting network topology can be regarded as an unstructured P2P system with peers deciding by themselves to which neighbors links and mappings are established. An interesting problem in this context is how to find the best set of neighbors for a new peer. In principle, a peer might use schema matching techniques [167] to find a peer whose schema is most similar to its own. It seems to be worthwhile to establish mappings to peers with similar data but also to establish some *long-range* links [159] to peers with different data. A detailed discussion on this aspect would be out of the scope of this dissertation. Thus, in the following we assume that such a policy to find appropriate neighbors exists and that a peer joining the network knows at least one peer that already participates.

On the logical level each peer P in a PDMS and its neighborhood can be regarded as a stand-alone data integration system with P serving as mediator defining the global schema. Figure 1.4 shows an example with highlighted data integration systems for peers P_0, P_9, and P_{10}. Because of the overlap of these integration systems, queries and data can be exchanged. In accordance to the definition of data integration systems given in [24, 119] we define a PDMS according to Definition 1.1.1.

Definition 1.1.1 (Peer Data Management System, Communication Link, Peer Schema, Source Schema, Mapping)**.** *A* Peer Data Management System $\mathcal{P}$ *is a pair* $\langle \mathcal{N}, \mathcal{I}_\mathcal{N} \rangle$*, where*

- $\mathcal{N}$ *is an arbitrary unstructured P2P overlay network that defines logical directed* communication links *of the form* (P_1, P_2) *to express the existence of a link from peer* P_1 *to peer* P_2 $(P_1, P_2 \in \mathcal{N})$
- $\mathcal{I}_\mathcal{N}$ *is the set of integration systems that contains exactly one element* $\mathcal{I}_P = \langle \mathcal{G}_P, \mathcal{S}_P, \mathcal{M}_P \rangle$ *for each* $P \in \mathcal{N}$*:*
  - $\mathcal{G}_P$ *is* P*'s* peer schema *expressed in a language* $\mathcal{L}_{\mathcal{G}_P}$ *over an alphabet* $\mathcal{A}_{\mathcal{G}_P}$*. The alphabet comprises a symbol for each element of* $\mathcal{G}_P$ *(i.e., for each relation if* $\mathcal{G}_P$ *is relational),*
  - $\mathcal{S}_P$ *represents the peer schemas of all of* P*'s neighbors* (source schemas)*, i.e., all peers that* $\mathcal{N}$ *defines a communication link to.* $\mathcal{S}_P$ *is expressed in a language* $\mathcal{L}_{\mathcal{S}_P}$ *over an alphabet* $\mathcal{A}_{\mathcal{S}_P}$*.* $\mathcal{A}_{\mathcal{S}_P}$ *comprises a symbol for each element of* $\mathcal{S}_P$*.*

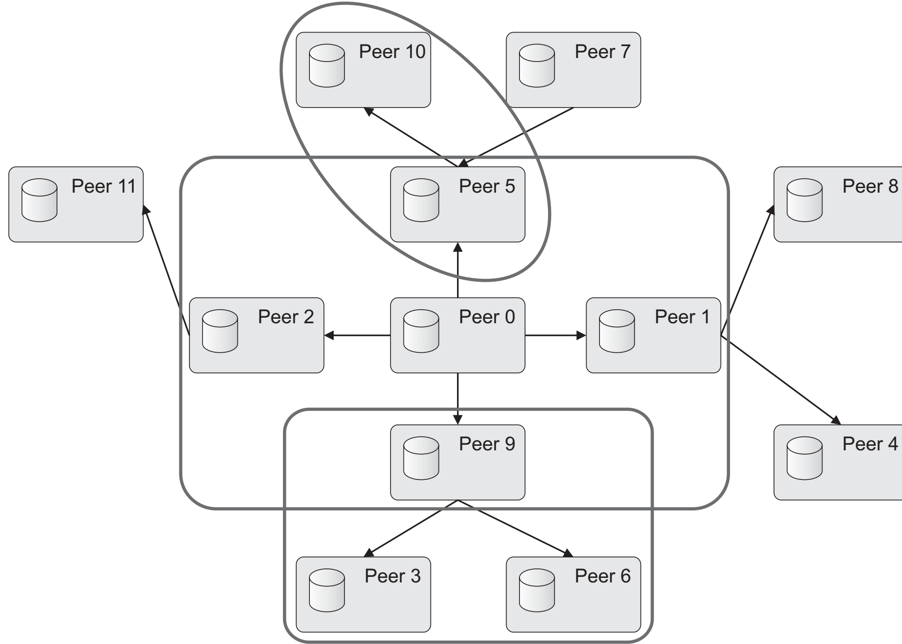


Figure 1.4: PDMS as a Combination of Multiple Data Integration Systems. *Highlighted data integration systems for peers* P_0*,* P_9*, and* P_{10}*.*

– $\mathcal{M}_P$ *is a* mapping *between* $\mathcal{G}_P$ *and* $\mathcal{S}_P$ *consisting of a set of assertions of the form*

$$q_{\mathcal{S}_P} \subseteq q_{\mathcal{G}_P}$$

where $q_{\mathcal{G}_P}$ *and* $q_{\mathcal{S}_P}$ *are two queries of the same arity over* $\mathcal{G}_P$ *using* $\mathcal{A}_{\mathcal{G}_P}$ *resp. over* $\mathcal{S}_P$ *using* $\mathcal{A}_{\mathcal{S}_P}$*.*

Whenever a query is issued in a PDMS, a peer uses its mappings to rewrite the query into the schemas of its neighbors. However, there are multiple possibilities to define such mappings. Their complexity mainly depends on the queries involved in the assertions. In this dissertation, we focus on the LAV (Local-As-View) approach, which means the assertions defining mappings (Definition 1.1.1) between peers are all of the same type: $s \subseteq q_{\mathcal{G}_\mathrm{P}}$ with $s \in \mathcal{S}_\mathrm{P}$ referring to a relation in the source schemas and $q_{\mathcal{G}_\mathrm{P}}$ being a query referring to relations in the global schema – Chapter 2 discusses this issue in more detail.

1.2 Distributed Query Processing in PDMSs

With respect to query processing in conventional distributed database systems [150], e.g., federated database systems, we distinguish between several steps (Figure 1.5). The first step is query formulation: a query needs to be formulated by the user or by a user application. The query has to adhere to a syntax that is understood by the query parser, which then translates the query into an internal representation. In general, the internal representation is a query plan which consists of logical operators describing the operations that need to be executed in order to compute the answer to the query.

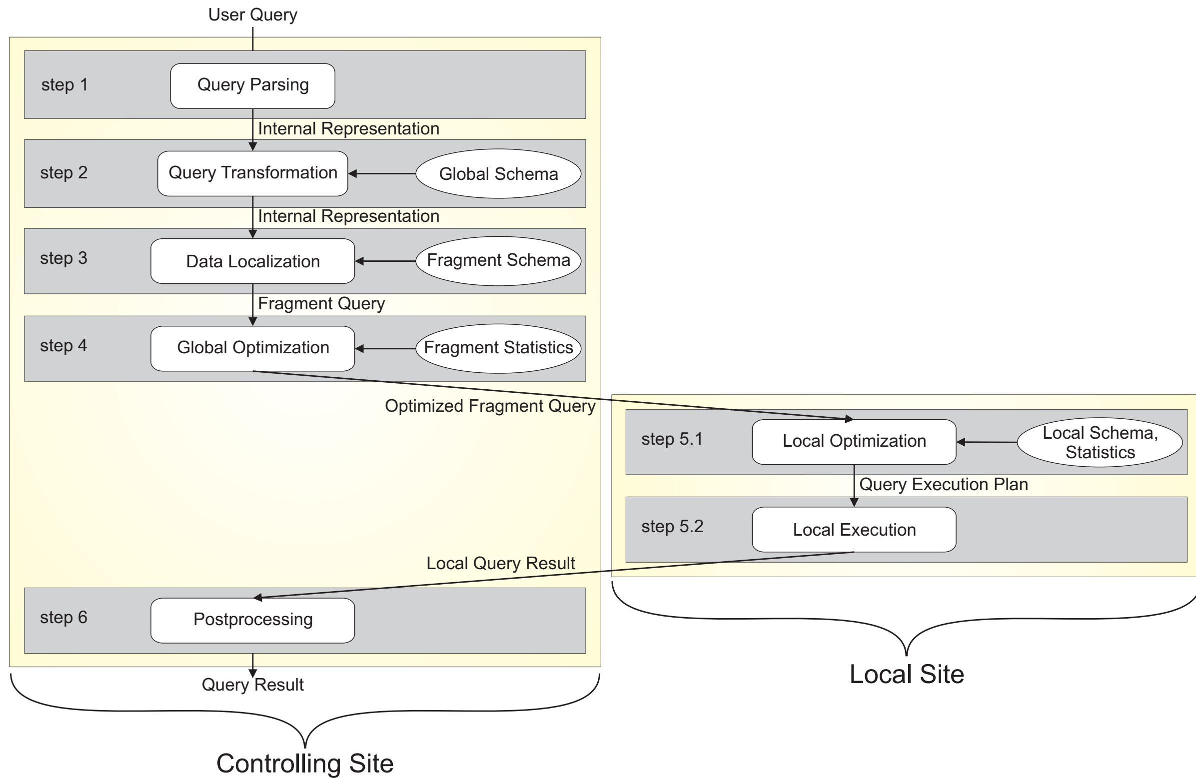


Figure 1.5: Distributed Query Processing

In the next step, the query is transformed in consideration of the global schema, which is available in distributed systems such as federated database systems. This includes the application of equivalence rules in order to effect normalization, unnesting, and simplification. The resulting query still references elements of the global schema. Thus, in the next step (data localization) the query is transformed into a fragment query that only references schema elements of the data sources. The initial fragment query is simplified by eliminating redundant subqueries and subqueries producing empty results. Afterwards, the fragment query is optimized (global optimization). In distributed systems such as federated database systems [40], the optimizer replaces logical operators with specific algorithms (plan operators) and determines the order of execution. In consideration of statistics, indexes, and the applied query processing strategy the optimizer also determines at which site an operation is to be executed and inserts corresponding communication primitives into the plan. The output of this step is a globally optimized fragment query formulated on the fragments. Parts of this query are executed nonlocally at participating nodes, which optimize their subqueries in the style of centralized database systems using local statistics. After the query has been executed, the partial results are sent to the controlling site. In most cases the postprocessing step, which is executed at the controlling site, consists of a simple union of the partial results received from the nodes. Finally, the result can be output to the user application.

As PDMSs are distributed systems, the general steps of query processing, as illustrated in Figure 1.6, are similar. The first step of query formulation and parsing is the same as for other distributed systems: the query is parsed and transformed into a query

plan consisting of logical plan operators that refer to schema elements of the initiator's schema and represent the operations that need to be executed in order to compute the answer to the query.

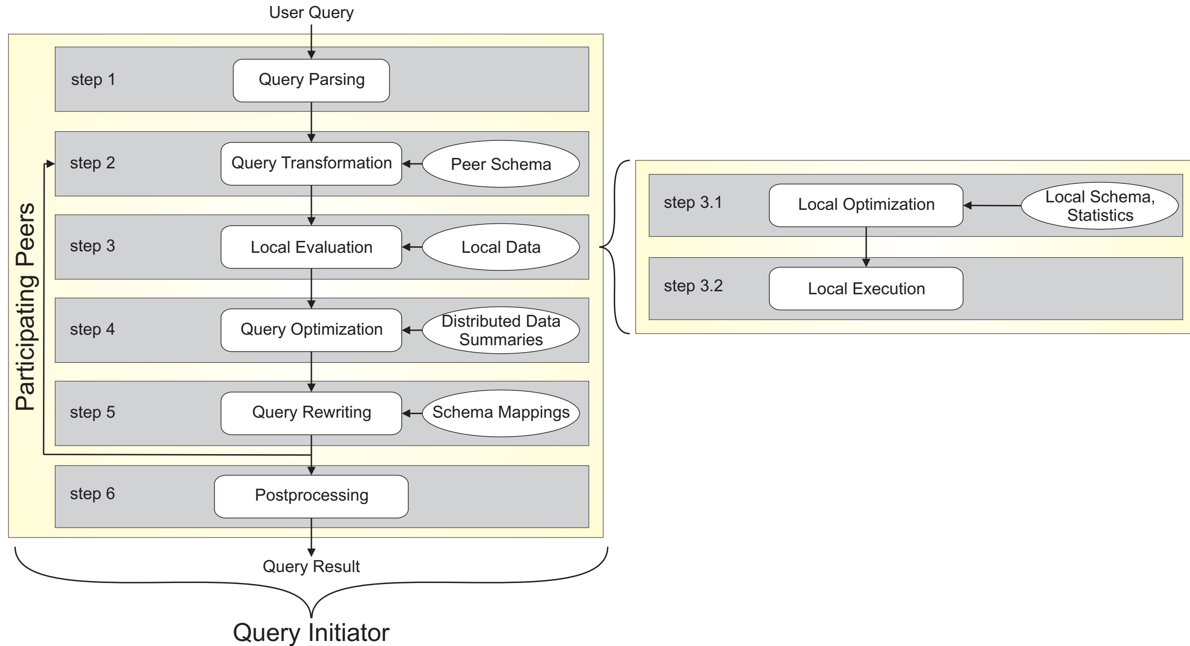


Figure 1.6: Query Processing in a PDMS

The second step (query transformation) again concerns the application of equivalence rules to effect normalization, unnesting, and simplification of the parsed query. In general, the initiator of the query itself provides local data that is relevant to answer the query. As already mentioned in Section 1.1, we assume that a peer's peer schema does not differ from the schema it uses to manage its local data. As the input query is formulated with respect to the initiator's peer schema, the query can be evaluated on its local data – applying optimization techniques in the style of centralized systems (step 3).

As a consequence of the absence of global knowledge, the optimizer (step 4) does neither know if there is data relevant to the query available in the network nor which particular peers in the system hold it. For this purpose, we propose the application of *Distributed Data Summaries* (DDSs) [90, 118, 213] – Chapter 5. DDSs are a subclass of routing indexes [41] providing summaries of the data that can be accessed by contacting particular neighbors. Although many PDMS implementations do not consider statistics or routing indexes for query optimization at all, we argue that it is an important step (step 4) that should be considered even before rewriting the query into the schema of neighboring peers.

In addition to DDSs the optimizer uses a set of rules that, optionally in conjunction with the output of step 3 (the partial result evaluated on the initiator's local data), helps to refine the query with respect to the data provided by the neighbors. These rules are defined by query processing strategies that consider the special characteristics of query operators, e.g., rank-aware operators. Furthermore, in order to minimize query execution costs, these strategies can also consider *approximation* techniques [92, 93, 213]. This kind of optimization has not yet been payed much attention to in PDMSs so that

the optimizations considered in this dissertation (Chapter 6) are novel in the context of PDMSs.

Based on the set of relevant neighbors and the refinements determined in step 4, the initiator uses query rewriting algorithms [94, 165] to create a query plan that contains subqueries formulated in the schema of neighboring peers. Therefore, a peer needs to know the correspondences between local schema elements and schema elements of its neighbors (provided by schema mappings). In this dissertation (Chapter 4), we propose an appropriate extension to existing algorithms that considers not only select-project-join queries but also rank-aware queries such as top-N and skyline.

After query optimization and rewriting, the subqueries are sent to neighboring peers. With the absence of global knowledge, which we assume a PDMS adheres to, the initiator of the query cannot optimize the query plan with respect to all the peers in the system – much in contrast to conventional data integration systems, where a central instance knows mappings to all data sources. Thus, global reasoning is impossible in this setup. As a consequence, each peer that receives a subquery from one of its neighbors needs to perform the same kind of optimization as the query's initiator. Hence, the peers receiving the subqueries process them by following steps 2 through 6 – possibly by contacting further peers in the system (chaining) – and send their results back to the initiator.

In order to obtain the final query result, it is often necessary to add a postprocessing step (step 6). In most cases the task of this step is merging the partial result evaluated locally on the local data and the results received from queried neighbors. However, post-processing might also perform further tasks such as identifying and removing duplicates. Finally, the result to the query can be output to the user application that has issued the query.

1.3 Outline of Dissertation

In summary, the main contributions of this dissertation concern:

- *Rank-Aware Query Operators and Approximation*: State-of-the-art peer data management systems are mostly based on conjunctive queries. As rank-aware operators such as top-N and skyline are not part of this paradigm, the systems do not support these operators. However, as argued above, those operators in particular have the potential to minimize query execution costs and should therefore be considered. In addition to these high-level query operators, approximation is another aspect that this dissertation introduces into PDMSs.

- *Query Processing Strategies*: Many PDMS variants rely on a simple flooding approach that routes queries to all the peers in the system. This is expensive, scales poorly, and is therefore not recommendable. Some systems introduced centralized indexes to identify peers that are likely to hold relevant data for a query. We argue that this impairs peer autonomy in an unacceptable fashion, introduces a single point of failure, and should therefore be avoided. In this dissertation, we propose advanced query processing strategies that apply a subclass of routing indexes to identify relevant peers. Furthermore, these strategies exploit the characteristics of

rank-aware query operators and approximation in order to prune additional peers from consideration.

- *Rewriting*: In order to access the local data of a neighboring peer, the query formulated at the initiator needs to be rewritten into the schema of the neighbor. Several query rewriting algorithms have been developed for conjunctive queries, mostly in conjunction with relational data. In this dissertation, we assume XML to be the native data format of peers. Thus, in order to process queries involving rank-aware operators and XML data, this dissertation presents an appropriate extension.

This dissertation is structured as follows. In Chapter 2 we define the terminology used in this dissertation and provide background information on the basic concepts our work builds upon, i.e., we discuss related work on rank-aware query operators, approximation, query processing strategies, query routing, and query rewriting.

Chapter 3 defines the specifics of the PDMS model we are working with. The subsequent chapters present our solutions to the problems discussed above. In detail, we propose (i) an algorithm for query rewriting in consideration of rank-aware query operators – Chapter 4, (ii) a novel subclass of routing indexes that is most beneficial for query routing in PDMSs as well as corresponding maintenance strategies in the presence of data updates – Chapter 5, and (iii) query processing strategies that efficiently process top-N and skyline queries using routing indexes and optionally applying approximation – Chapter 6.

Chapter 7 presents SmurfPDMS (SiMUlating enviRonment For PDMS), a system developed in the context of this dissertation, which implements all algorithms proposed in the previous chapters. It also represents the platform we used to perform the evaluation of all proposed algorithms.

Finally, Chapter 8 concludes this dissertation with a summary and an outlook to future work.

Chapter 2

Background

In this chapter, we review related work that is relevant to the contribution of this dissertation. We provide a classification and discuss why existing strategies do not solve the problems we aim to solve. Following a bottom-up perspective on the system architecture, we begin with rank-aware query operators and approximation and proceed with query routing. Afterwards, we review related work on query processing strategies in distributed environments and query rewriting. We conclude with related work on PDMS implementations.

2.1 Rank-Aware Query Operators

Rank-aware queries are a very popular approach to deal with the possibly huge number of result records that the complete answer to a query might consist of. In the following, we focus on two representatives: *top-N* and *skyline*.

2.1.1 Top-N Queries

All rank-aware queries are based on ranking functions by definition. Top-N queries (Definition 2.1.1) use only one such function r in combination with an integer number N to limit the result size to those N records with the highest (lowest) ranks. In accordance to Börzsönyi et al. [19] we assume that each ranking function has an annotation (MIN or MAX) that indicates whether the highest or lowest ranked records are relevant. Ranking functions can be defined on a single attribute, e.g., ranking hotels according to their distance to the beach. But it is also possible to define them on multiple attributes, e.g., ranking hotels according to their distance to the beach and the price per night: $dist \cdot 0.5 + price$. The latter is an example of a ranking function using weights to indicate the importance of multiple attributes.

Definition 2.1.1 (Top-N Query). *Given an integer number N, a ranking function r, an annotation $\gamma \in \{MIN, MAX\}$, and a set of records D, then a top-N query $\mathfrak{T}_r^N$ on a set of records D is defined by function $t(N, r, \gamma, D) := \{d_1, d_2, \ldots, d_N\}$. The result set consists of those N records of D that are ranked highest ($\gamma = MAX$) resp. lowest ($\gamma = MIN$) by ranking function r.*

Figure 2.1 shows an example top-N query with $N = 10$, which ranks objects in two-dimensional Euclidean space according to their Euclidean Distance to a given reference record q – represented by the asterisk. More precisely, the ranking function is $\sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}$ with p denoting a data record in the input data set. The result consists of those N records with the minimum distances. The circle indicates the distance of the record with the maximum distance to the asterisk that is still part of the result set.

An important issue we want to stress with respect to processing top-N queries is whether the user is allowed to define arbitrary ranking functions or if the system poses any restrictions on their formulation. In this dissertation, we are working with arbitrary user-defined ranking functions (Definition 2.1.2) that are possibly unique to any query being issued so that the user is not restricted to a set of predefined ranking functions.

Definition 2.1.2 (User-Defined Ranking Function)**.** *A ranking function r is said to be* user-defined *if the user was free to compose r to his/her own discretion, limited only with respect to the set of arithmetic operations supported by the system.*

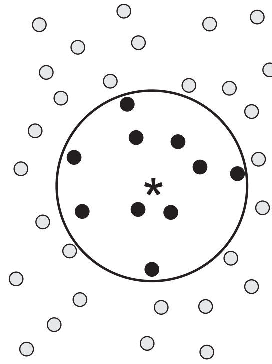


Figure 2.1: Top-N Query example

We can distinguish two classes of top-N query processing strategies: (i) those developed for centralized systems with global knowledge being available and (ii) those being applied in distributed environments.

Computing Top-N Queries in Centralized Systems

Since most users are not interested in hundreds of result records, which would be the exact answer to a query, it is a good idea to present the best N records ranked according to a user-defined ranking function. In this sense top-N queries have already played an important role in centralized relational database management systems (RDBMSs) before their introduction into distributed environments. Thus, several approaches and efficient strategies for top-N query processing have been developed. Chaudhuri et al. [23, 36], for instance, use existing data structures in RDBMSs such as indexes and histograms to map a top-N query into a traditional selection query by trying to determine a minimal n-dimensional rectangle that contains all the top-N records for a given query but as few additional records as possible. Another idea is to use materialized views to speed up top-N query processing [96, 208].

Computing Top-N Queries in Distributed Environments

Later on, when distributed systems gained more importance, algorithms such as Three Phase Uniform Threshold (TPUT) [27] have been developed. TPUT actually is an enhanced version of the Threshold Algorithm (TA), which was developed independently by multiple groups [82,131,146], but is often referred to as Fagin's threshold algorithm [131]. The aim of TPUT is to reduce network bandwidth consumption by pruning ineligible objects (data records). Each data source provides one attribute for a data object, i.e., data is distributed vertically among peers. The peers provide lists of objects sorted in ascending order of the respective attribute value. A central manager that has direct access to all peers coordinates query processing. In phase 1 each source is asked to send the top N items from its list to the central manager (sorted access). Based on the received information the manager computes partial aggregations (partial sums) for the objects with the present information. The partial aggregate value serves as a lower-bound estimate – unknown attribute values are assumed to be 0, which is the lowest possible attribute value. After the manager has ranked the objects, the N highest partial sum is used as a lower bound, denoted as τ_1.

In phase 2 the manager sets $T = (\tau_1/m)$ as threshold (with m being the number of attributes) and sends it to all peers. Each peer then sends the list of objects whose values are at least equal to T. If an object is not reported by any peer, then all its attribute values are smaller than T. This means that its aggregate value must be smaller than τ_1. Hence, it cannot be a member of the top-N result set and can be pruned. With the retrieved information the manager refines the lower bound estimate T by recalculating the partial sums for the objects, again by determining the N highest partial sum τ_2. This time further objects can be pruned using an upper bound estimate. To define the upper bound estimate, the manager computes an upper bound for each object. It is defined as the sum of all known attribute values and T for all unknown attributes values. We can safely do that because if the unknown attribute values were higher than T, they would have already been reported. Objects whose upper bounds are smaller than τ_2 are pruned.

In phase 3 the missing attribute values of the remaining objects, which have not been eliminated in step 2, are requested from the sources (random access). The manager can now calculate the complete aggregate value for each object, rank the objects accordingly, and output the top N objects to the user.

Cao et al. [27] also briefly discuss how to apply TPUT on hierarchical and P2P networks. The algorithm needs three round-trips, i.e., for answering a single query the network is flooded three times. The basic threshold algorithm [82,131,146] has also been adapted to situations where some of the lists may not be accessed in sorted access [132]. The key concept is to use the lists that can be accessed in sorted order in the same way as explained above, retrieve for each object the missing attribute values by random access, and compute the threshold using the value of 1 for those attribute values that can only be accessed in random order. However, further variants of TA have been developed for instance for multimedia repositories [37], distributed preference queries over web-accessible databases [133], and ranking query results on structured databases [7,35].

Another algorithm for processing top-N queries in distributed environments, i.e., in Edutella [145] P2P networks, is presented by Balke et al. [11]. This approach tries to improve top-N query processing by dynamically collecting query statistics that allow for

a better routing when the *same* query is issued the next time. The first time, however, all peers have to participate in processing the query while several round-trips are required in order to retrieve the final result.

Only recently, skyline-based techniques for processing top-k queries in super peer based P2P networks [197] have been proposed. The solution is based on the K-skyband [141,153] that is created in a preprocessing step at each super peer. Thus, each super peer maintains a K-skyband of all assigned peers which can be used to answer any incoming top-k query. Furthermore, peers exchange skylines (Section 2.1.2) to build up an index which can be used for query routing.

Finally, let us mention that top-N algorithms have also been developed for other application areas such as stream processing and information retrieval, e.g., [28, 141] to mention a few recent ones.

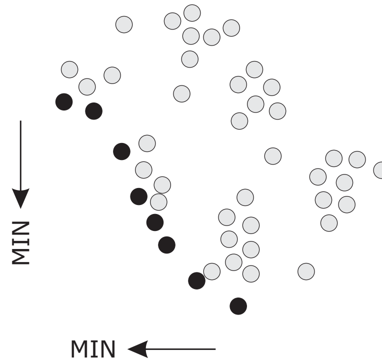


Figure 2.2: Skyline Query Example

2.1.2 Skyline Queries

Skyline queries can be considered a multidimensional variant of top-N queries since data records are not ranked according to one single ranking function but according to at least two of them. Let us first illustrate the principle with an example. Imagine a user is looking for hotels that fulfill two criteria: they should be as cheap as possible and as close as possible to the beach. In most situations it is not obvious whether the user would prefer (i) a hotel that is very close to the beach but not as cheap as others, or (ii) a hotel that is very cheap but farther away from the beach. Thus, it is important to present all "interesting" answers that might fulfill the user's preferences so that he or she can choose the most promising one. This set of interesting answers is called the skyline. Figure 2.2 shows a visualization of the example with each point representing a hotel and the axes corresponding to distance and price. Definition 2.1.3 formally defines a skyline.

Definition 2.1.3 (Skyline Query, Dominance Relation)**.** *Given a set of ranking functions $\mathcal{R}$ each annotated with $\gamma \in \{MIN, MAX\}$ and a set of data records D, then the skyline $\mathfrak{S}_{\mathcal{R}}(D)$ comprises all those data records that are not* dominated *by any other record. One data record p dominates another one q with respect to $\mathcal{R}$ ($p \prec_{\mathcal{R}} q$) if it is as good as or better in all dimensions and better in at least one dimension, that is:*

$$p \prec_{\mathcal{R}} q :\Leftrightarrow \begin{array}{ll} (i) & \forall r \in \mathcal{R} : r(p)\ \Theta_{eq}(\gamma)\ r(q),\ \textit{and} \\ (ii) & \exists r \in \mathcal{R} : r(p)\ \Theta(\gamma)\ r(q) \end{array}$$

and

$$\mathfrak{S}_{\mathcal{R}}(D) := \{d \in D \,|\, \nexists p \in D : p \prec_{\mathcal{R}} d\}$$

Θ_{eq} and Θ denote the comparatives in dependence on the annotation:

$$\begin{array}{lll} \Theta_{eq}(MIN) & \rightarrow & \leq \\ \Theta_{eq}(MAX) & \rightarrow & \geq \\ \Theta(MIN) & \rightarrow & < \\ \Theta(MAX) & \rightarrow & > \end{array}$$

Two data records p and q are said to be incomparable *if $p \nprec_{\mathcal{R}} q$ and $q \nprec_{\mathcal{R}} p$.*

In the example introduced above (Figure 2.2) object o_1 is for instance dominated by an object o_2 if o_2 is cheaper and located at least as close to the beach as o_1. Each pair of objects that is part of the skyline is incomparable.

With respect to computing skylines we again distinguish between two main classes of algorithms: (i) computation in centralized systems and (ii) computation in distributed systems. Moreover, we identify another class that comprises skyline variants as extensions of the basic concept. Figure 2.3 illustrates the two main classes and their subclasses that we discuss in the following.

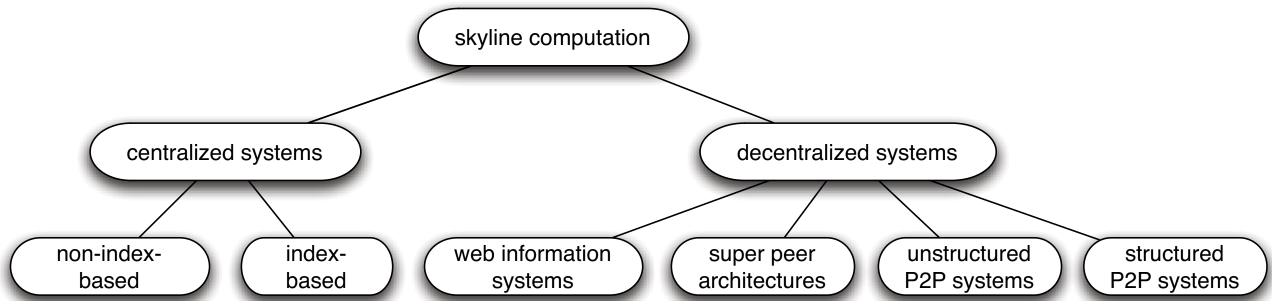


Figure 2.3: Classification of Algorithms for Skyline Computation

Computing Skylines in Centralized Systems

Already before the introduction of skylines into database research [19], the same problem had been known as the Pareto optimum or the maximum vector problem [117, 164]. Along with a corresponding SQL extension, allowing MIN, MAX, and DIFF as annotations, Börzsönyi et al. [19] present several basic main memory algorithms for computing the skyline in centralized databases: BNL (Block Nested Loops) and D&C (Divide & Conquer). The principle of both algorithms is that instead of comparing each data record to all the others, blocks of records are compared. BNL maintains a set of records in main memory (window w). Records are read one after another. Each record p is checked for dominance by any record that is currently contained in w. If there is a record q in w that dominates p ($q \prec_{\mathcal{R}} p$), then p can safely be discarded as it cannot be part of the skyline. If there is no such record q, p is inserted into w. Finally, each record in w that is dominated by p is removed. As the size of w is limited, there is a swap file.

Whenever w is full and a record shall be inserted into w, the record is written into the swap file. After all records of the input set have been read, those records in w that have been compared to all the others can be output as skyline records and removed from w. Then, the records of the swap file are checked for dominance as explained above. These steps might have to be repeated several times, but finally all skyline records are found.

Applying the divide and conquer algorithm (D&C), the set of input records is recursively divided into smaller sets. By exploiting the additivity characteristic of skylines, i.e., $\mathfrak{S}_{\mathcal{R}}(\mathfrak{S}_{\mathcal{R}}(A) \cup \mathfrak{S}_{\mathcal{R}}(B)) = \mathfrak{S}_{\mathcal{R}}(A \cup B)$, the skyline is computed in the smallest partitions first. Then, the skylines computed for these partitions are merged recursively until finally the skyline on the complete input set can be determined. Another important observation that D&C exploits is illustrated in Figure 2.4: regions $S_{1,2}$ and $S_{2,1}$ do not need to be checked for mutual dominance at all since it is impossible that records in these regions dominate each other, i.e., all records in $S_{1,2}$ are incomparable to records in $S_{2,1}$ and vice versa. Börzsönyi et al. [19] also sketch approaches which exploit sorted indexes (B-tree and R-tree) proposing an extension to Fagin's A_0 algorithm [58] for use with B-Trees and the branch-and-bound principle for use with R-trees. However, both approaches have been elaborated on in future work and are discussed below.

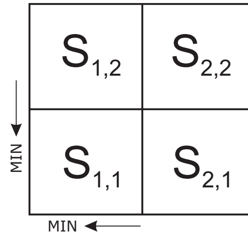


Figure 2.4: D&C Dominating Regions

In order to speed up processing time, index-based algorithms (e.g., [183] using precomputed bitmap and B-tree indexes) have been proposed. They describe each attribute of each single data record as bitmaps. Assume we have two dimensions (attributes) with 10 distinct attribute values each. In that case, each record would be represented by a 20 bit long vector (10 per dimension, i.e., one bit per distinct value). A data record p is encoded as follows: for each dimension the bit corresponding to p's attribute value is set to 1, i.e., if p's attribute value is the second "best" of all existing values, the second bit is set to 1. All bits that represent "worse" values are also set to 1 while all other bits are set to 0. This is done for all data records and their attribute values in the input data set. In order to determine if a data record p belongs to the skyline, i.e., to determine whether there is a record q that dominates p, all that needs to be done is to compare the bit vectors. Remember the definition of the dominance relation (Definition 2.1.3). If a record q dominates p, then (i) all of q's attribute values are at least as good as those of p and (ii) at least one attribute value of q is better than the corresponding value of p. Consequently, all records q that might dominate p must have at least as many 1's in their bit vectors for each dimension as p – if they have exactly the same bit vector representation in one dimension the corresponding attribute values are the same. If q has only one more 1 in the bit vector for a dimension, this means that q's attribute value in that dimension is better than p's.

The second approach presented by Tan et al. [183] uses a B-tree to index data records. There is one B-tree index for each dimension (attribute). Each data record p is inserted into the index corresponding to p's minimal attribute value. Consequently, each record is contained in only one of the indexes. For instance, if all records have two attributes and $p = (1, 5)$, then p is inserted into the index corresponding to the first attribute. Assuming we want to compute a MIN skyline, then the indexes are sorted in ascending order. The algorithm treats the indexes as sorted lists of batches, a batch is defined as the set of all records in an index with the same attribute values. The skyline computation starts with the list that contains elements with the lowest attribute value. All records within the corresponding batch are checked for dominance. The algorithm proceeds with the list providing the lowest not yet processed attribute value. Again, all records of the corresponding batch are checked for dominance. This goes on until one batch contains a record p whose attribute values are smaller than all those batches of all lists which have already been processed – p dominates all records that are not yet read.

Algorithms such as NN-search (Nearest Neighbor) [116] and BBS (Branch and Bound Skyline – using an R-tree) [152] have also been developed for application in centralized databases. Their main advantage is that they can progressively give an overview describing the final result already after a short time of processing. The main principle of these algorithms is to compute the nearest neighbor to the origin (minimizing all ranking functions). The nearest neighbor is guaranteed to be part of the result skyline and output to the user because there is no data record that might dominate it. Then, the nearest neighbor can be used to partition the data space. Obviously, the region dominated by the nearest neighbor does not have to be considered any further and can safely be pruned from consideration. By looking recursively for the nearest neighbor in each of the remaining partitions, the total skyline can be determined.

An extension of BBS supports approximate skylines [153], i.e., skylines containing hypothetical points. Based on the regions of the R-tree index hypothetical points are probabilistically determined and a so-called "low-resolution" skyline is output to the user. This enables a "drill-down" exploration of the actual skyline, which is refined iteratively by traversing the R-tree index from root to leaf nodes.

However, as recent publications show [12, 39, 71], research has not yet solved this problem, not even for centralized systems. For more details we refer to a recent survey on skyline computation in centralized systems [72].

Skyline Variants and Derivatives

In recent years, new applications and variants of skylines have been proposed, for instance the k most representative skyline operator (top-k RSP – top k representative skyline points) [130]. Top-k RSP is defined as the k-element subset s of the skyline that of all possible k-element subsets dominates the largest number of data records, i.e., s maximizes the set of dominated data records.

Another variant of skylines are multi-source skylines [48,176]. Whereas a single-source skyline query might ask for hotels that are cheap and close to the beach (or close to any given location), a multi-source skyline query considers multiple records at the same time (e.g., to find hotels that are cheap and close to the university, the botanic garden, and china town). The algorithms presented by Sharifzadeh et al. [176] deal with the problem

in Euclidean space whereas Deng et al. [48] propose algorithms for use in road networks. In these networks it is impossible to compute the distance between any two records using the Euclidean Distance.

Further variants of skylines are constrained skylines [152], subspace skylines [185, 186], and skycubes [155, 157, 204, 211]. A constrained skyline query returns the most interesting records in the data space defined by a set of constraints. Each constraint defines a range along a dimension (attribute) and the conjunction of all constraints forms a hyper-rectangle in d-dimensional space. For example, looking for hotels that are cheap and close to the beach a user might define a constraint for the price, e.g., restricting the price to $\$100-\$300$ in order to guarantee a certain standard. Figure 2.5 illustrates this principle – the region defined by the constraints is highlighted in grey. In comparison to the non-constrained skyline (black circles), the constrained skyline (red circles) only consists of records that are contained in the region defined by the constraints. In case of a subspace skyline we do not restrict the attribute values but the set of attributes that are considered for computation. For instance, in addition to price and the distance to the beach the hotel data records could additionally possess an attribute indicating a user-ranking. Now, computing a skyline on only two of the three dimensions conforms to computing a subspace skyline. There are even some approaches which combine constrained and subspace skylines [46]. Another variant that is worth mentioning is the skycube, which is defined as the complete computation of all possible subspace skylines – similar to the idea of the data cube [6] for data warehouses.

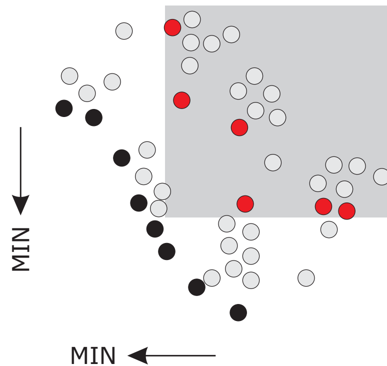


Figure 2.5: Constrained Skyline Query Example

Some works focus on skylines in high-dimensional spaces, i.e., a skyline defined on a high number of attributes/ranking functions. In such spaces the result set of a skyline query consists of many records. In order to give the user a better overview, Chan et al. [32] propose to select only the top k records with the highest skyline frequency, i.e., the top k records that are contained in the skyline of as many subspaces as possible. As the computation of the top-k frequent skyline records would require to compute the skyline in each possible subspace, Chan et al. propose an approximate algorithm which trades off accuracy for query response time. A related concept, the k-dominant skyline, is proposed by Chan et al. in [31]. To output only the most interesting skyline records in high-dimensional space, the dominance relation is relaxed to k-dominance. A record p k-dominates another record q if p is as good as or better in k dimensions and better in at least one of these k dimensions.

Spatial skyline queries (SSQs) [176] are skylines considering the spatial distance between records. For this kind of queries the dominance relation is adapted so that a record p spatially dominates p' with respect to database D if every $q_i \in D$ is closer to p than to p'. Yet another variant of skyline queries, dealing also with spatial attributes, is represented by neighborhood dominant queries (NHDQs) [126]. They were developed to support a micro-economic approach towards decision making. The key concept is to consider not only the dominance relationship between records but also their spatial distance to each other. Given hotel data (quality, price, coordinates x and y, etc.) and a specific hotel p, an example for such a query would be to find the nearest hotel q that dominates p with respect to quality and price. To compute such queries efficiently, Li et al. propose index-based methods (R-trees and bitmaps).

In general, skyline computation is considered for totally-ordered attribute domains. However, there are a few approaches towards skyline computation for partially-ordered attribute domains. An example for such data is categorical or set-valued data. Chan et al. [29, 30] present a technique that transforms partially-ordered attributes into two integer-domain attributes and organizes the data by using an index. Apart from two further variants, Chan et al. propose an advanced version of the Branch and Bound Skyline algorithm (BBS) to compute the skyline on the transformed data. Chaudhuri et al. [34] study cardinality and cost estimation of the skyline operator in relational engines involving partially ordered attributes. Wong et al. [201] go one step beyond partially-ordered attributes: nominal attributes. These are categorical attributes that do not have a fixed predefined order: as different users might have different preferences on nominal attributes, more than one order needs to be considered.

Jin et al. [104] propose thick skylines, i.e., not only the skyline records are output to the user but also all records within an ε-distance around them. The intention is to give the user a choice between several almost equally good records instead of outputting only the best one that dominates the others. Jin et al. propose several methods to compute such skylines within RDBMSs based on sampling and indexing the data.

Only recently another variant of skylines has been proposed: reverse skylines [56]. Given a point q, all other points are represented by their distance vector to q. A skyline on this data is also called a dynamic skyline. The reverse skyline query returns the objects whose dynamic skylines contain the query object q. To compute such queries, Dellis et al. propose an improved and adapted version of the BBS algorithm. In order to speed up query processing, Dellis et al. use precomputed approximations of skylines.

Materialized views are an effective possibility to reduce query response time, which has been used in various contexts for many years [84]. The same principle can be applied to skyline query processing. However, the problem is how to maintain materialized skyline views in the presence of data updates (insertions and deletions). The problem of skyline maintenance has already been considered in conjunction with the BBS algorithm [153]. The addition of records is straightforward and can be solved efficiently by considering only the records contained in the view, i.e., in the current skyline. However, the deletion of skyline records is more complicated and requires reevaluating records of the original data set, as they might be part of the skyline now. Papadias et al. [153] propose to issue constrained skyline queries on the exclusive dominance region of the deleted record. This is the region in data space that is only dominated by the deleted record. As this idea is not much elaborated on by Papadias et al. [153], Wu et al. [202] elaborate on this idea

and examine the shape of such exclusive dominance regions. Furthermore, they propose an advanced algorithm to update skylines in the presence of deletions.

Finally, let us mention that skyline query processing is also considered in further application areas such as data stream mining [129, 140, 184, 191, 205]. Even continuous skyline queries for moving objects, i.e., data values dependent on space and time, have been considered [98]. Furthermore, the problem of processing skylines has also been considered for uncertain data (probabilistic skylines) [156] and incomplete data [110].

Skylines in Distributed Environments

Most work on skyline queries has been designed for use in centralized systems. There are only a few ones that consider computing skylines in distributed environments. However, most of these few are restricted to specialized network structures, which makes them hardly applicable to the general case or unstructured P2P systems.

One of these algorithms [10] exploits the TA principle of sorted lists, which we have already discussed in the context of top-N queries above, for processing skylines in Web Information Systems. Again, the setup assumes data to be distributed vertically across different Web Information Services with each site providing one attribute of a data record. Each web source provides a score list in globally sorted order. The principle of the basic algorithm is the same as for computing top-N queries: in step 1 sorted access is used on all lists until the same object has been read in all lists, in step 2 additional sorted access is performed to retrieve objects with the same attribute values, and in step 3 random access is used to retrieve the attribute values of all objects that have been read so far in any list. Finally, all objects are checked for dominance and the result is output to the user.

Another work considering skyline computation in distributed environments is DSL [203]. It progressively computes constrained skylines using a CAN [168] network to route queries through the network. Each peer is assigned a region of the data space. There are two main principles: first, peers can work in parallel if none of their records can dominate each other. Second, due to the data organization in CAN networks, all regions S_R and corresponding peers can be pruned from consideration once a peer has been contacted whose assigned data region dominates all regions in S_R.

Wang et al. [198] present a strategy (Skyline Space Partitioning, SSP in short) for distributed processing of skyline queries in BATON [101] networks. Peers in BATON are organized in a balanced tree structured P2P overlay network that can be used to route queries efficiently. In such a network each peer is responsible for a data region. Techniques for splitting and merging allow load balancing among peers. By dynamically sampling load from random nodes, load imbalance can be detected and data may be migrated to other peers in order to counteract the imbalance. The principle of processing skylines is straightforward: identify regions that are not dominated and send the query to peers holding data of those regions using the BATON structure to route the queries. More precisely, looking for a $MIN - MIN$ skyline on a two-dimensional data set, skyline computation starts at that peer p_{start} whose local results are guaranteed to be in the final skyline (i.e., the peer being responsible for the region containing the origin). This peer computes the local skyline and selects the most dominating record p_{md} (i.e., the record dominating the largest region [97]). Then, p_{md} is used to refine the search space

and thus to prune irrelevant regions and peers from consideration. Wang et al. state that this is an important difference to the approach proposed by Wu et al. [203], which uses all intermediate skyline results for this purpose. A peer can safely be pruned with respect to the skyline definition if its region's best record is dominated by p_{md}.

Another work [127] exploiting the same principle on top of structured overlay networks uses an overlay network called Semantic Small World (SSW) to organize peers. Peers are organized in clusters based on semantic labels that are centroids of their largest data clusters. Each peer is assigned to a cluster, which is defined by a region in data space. Each peer maintains information about peers providing data in the same data space. A peer maintains short-range links to peers within the same cluster and some long-range links to peers of other clusters. The computation of a skyline starts in the cluster that is guaranteed to contain skyline records, i.e., in case of a MIN-skyline the cluster containing the origin. Then, the skyline in this cluster is computed. The corresponding result can already be output to the user because it is guaranteed to be in the final result as we have already discussed above (NN-search). The nearest neighbor to the origin is used to determine those clusters that still have to be considered – pruning all clusters that are dominated by the nearest neighbor. Finally, the partial skylines are combined and the result can be output to the user.

Apart from this exact algorithm Li et al. [127] also propose an approximate algorithm, which can be used when it is impossible to construct an SSW overlay network. Still, peers have semantic labels that can be used to process the skyline. As a peer knows the semantic label of its neighbors, it forwards the query to the neighbor that has a "better" label with respect to the query definition (single-path). This means the query is sent to a peer providing data which might be ranked better than the local data. Once a peer receives the query twice, skyline computation ends and the result is output to the user. A second variant of this concept (multi-path) considers each queried attribute in parallel and forwards the query to each peer providing "better" data for at least one attribute. This variant makes heavy use of the peer that initiates the query because it has to coordinate the computation and issue a stop command when a certain number of peers has been queried. The application of these approximate algorithms makes only sense if the data provided by each peer is clustered. When data is distributed randomly, the chance of missing relevant data is high. Furthermore, the use of the base algorithm creates heavy load in the cluster containing the origin, which might become the bottleneck of the system if skyline queries are issued often.

Vlachou et al. [196] propose an algorithm for processing skylines in super peer networks. In these networks some peers (super peers) are assigned special roles due to their enhanced features such as availability, storage capacity, and bandwidth capacity. In the baseline algorithm each super peer holds an extended skyline on the data of all its subordinate peers. The extended skyline is the subset of the original data set containing the skyline records of any possible subspace. Storing such an extended skyline at each super peer reduces execution costs by not having to ask every single peer in the system but only all super peers. Query processing takes place by flooding the backbone network of super peers. Each super peer computes a partial skyline and sends additional information along with the query to its neighbors when flooding the network.

This additional information is a consequence of the application of the following observation: multi-dimensional data records p can be transformed into one dimensional values

by the following formula: $f(p) = \min_{i=1}^{d}(p[i])$. Let $dist_U(p)$ denote the L_∞-distance of record p from the origin based on the dimension set U, i.e., $dist_U(p) = \max_{i \in U}(p[i])$. A record p for which the following inequality holds cannot be in the skyline of subspace U: $f(p) > dist_U(p_{sky})$ with p_{sky} being a skyline record in subspace U. In other words, when the minimum coordinate (all dimensions) of a record p is higher than the maximum coordinate (queried dimensions) of a skyline record p_{sky}, then p cannot be part of the skyline. Exploiting this observation the additional information, which a peer attaches to a query when forwarding it, consists of the minimum value of $dist_U(p)$ of all records p that are part of its locally computed partial skyline. The receiver can use this value to prune all records with higher $f(p)$ values than the value received along with the query because they are dominated by already known records.

Very recently Cui et al. [43] proposed an algorithm for processing constrained skyline queries in unstructured P2P networks. When a query is issued at a peer, it first collects statistics from all the peers in the network, i.e., for each peer the d-dimensional bounding box of its local data is retrieved – assuming that each peer provides data only with respect to a certain cluster in data space. The main principle is to determine which peers can process the query in parallel. Thus, the initiator determines, based on the dominance relation and the global knowledge in the form of the minimal bounding boxes, peer groups whose results cannot dominate each other. For each of these groups the initiator determines a query plan, which is sent to one peer of each group. Once a peer has processed the query, it removes itself from the query plan and forwards the query to the next peer indicated by the plan. In order to reduce network traffic, each peer attaches some filter records to the query. These are data records that are likely to dominate many other records peers might store locally. The results are first collected per group at the designated peers and then sent to the initiator.

2.1.3 Applicability to PDMSs

Most of the approaches towards top-N and skyline computation we have discussed are not applicable to PDMSs. Those developed for centralized systems have been designed to reduce main memory consumption as well as IO and CPU costs. Since in P2P systems data is distributed over many peers, the only way to apply most of these approaches would be collecting all the data at the initiating peer and computing the query there. Obviously, this is not an acceptable solution. However, some of the approaches for centralized systems use indexes to accelerate computation. Note that this kind of indexes is of severely limited use in unstructured P2P networks and PDMSs as it requires indexing all available objects. With the absence of global knowledge it is hard to provide direct access to the sources and such detailed indexes. Of course, we also propose the use of indexes to enable efficient query routing but these semantic routing indexes describe data records with as few values as possible trying to reduce memory consumption and maintenance costs by applying summarization/approximation techniques. Most important is the fact that routing indexes are general purpose and not restricted to any special kind of queries, i.e., they are useful for any kind of queries: top-N, skyline, selection, etc.

We have also discussed several other approaches developed for distributed systems. However, the problem is that they strongly depend on a specialized network structure. One of these algorithms is TA and all its variants and advanced versions for top-N and

skyline query processing. All these algorithms based on TA cannot be applied to PDMSs because the algorithm requires lists of objects in globally sorted order. Furthermore, the data distribution in PDMSs is totally different; a peer does not provide only one attribute but is likely to provide all attributes for a particular object. Finally, there is no central manager with global knowledge which is able to directly contact all participating peers. However, Cao et al. [27] nevertheless briefly discuss how to apply TPUT in hierarchical networks and P2P networks. The algorithm still needs three round-trips, i.e., for answering a single query the network is flooded three times. Although the algorithm might still work in P2P systems, we argue that flooding a network consisting of more than only a few peers should strictly be avoided because flooding the network, even only once, is in most cases too expensive. Thus, flooding it multiple times to answer a single query cannot be considered an efficient solution.

We have also sketched several algorithms developed for use in structured P2P overlay networks. These algorithms are efficient in the networks they have been developed for because the specialized network topology can be used to route queries efficiently. On the other hand, they cannot be applied to PDMSs for the same reason. Trying to preserve peer autonomy, PDMSs do not impose any network structure upon the network. Most importantly, data in PDMSs is not redistributed among the peers in the network such that the routing and pruning algorithms proposed by algorithms developed for structured P2P systems cannot be used in PDMS.

We have also discussed an approach for processing skyline queries in super peer systems [196]. The backbone network consisting of the super peers could be considered an unstructured P2P system. However, in order to process a query, it is forwarded to all the super peers in the network, i.e., applying a simple flooding approach. Although the number of super peers is much lower than the total number of participating peers, this approach is still likely to be expensive and to cause high network and computational load.

The strategy for processing skyline queries recently proposed by Cui et al. [43] was designed for application in unstructured P2P systems. However, although PDMSs are built upon such systems, the proposed strategy is based on global knowledge available at the initiator's site in order to determine peer groups and query plans. Furthermore, each peer is assumed to provide data only with respect to a cluster in data space.

Finally, let us comment on the techniques proposed by Balke et al. [11]. Dynamically collecting query statistics is an interesting approach but only worthwhile if the same query is issued multiple times. The first time, however, all peers in the system need to participate. In this dissertation, we try to find a solution that does not rely on the user issuing the same query twice but tries to be optimal even when queries with completely unknown ranking functions are issued. On top of these algorithms we can still use caching in conjunction with algorithms for testing query containment such that results from previous queries can be used to accelerate the computation of future ones.

2.2 Approximation

Approximation is a very popular approach to reduce execution costs. There are different types of approximation; all of them have two things in common. First, the algorithms

need parameters as input limiting the degree of applicable approximation (in most cases defined by the user). Second, approaches applying approximation need to provide some kind of guarantee which estimates the goodness of the result. Usually, the guarantee is a simple consequence of the input parameters that limit approximation. An example of such guarantees are probabilistic guarantees as proposed in the context of top-N queries [89, 190].

As this dissertation is focusing on skyline and top-N queries, the following pages discuss the classes of approximation the literature proposes for these queries. In this context, we identify the two classes illustrated in Figure 2.6. The first kind of approximation is pruning and introduced by the query processing strategy, i.e., the data remains untouched/unapproximated but for example the algorithm stops searching for further result candidates at an early stage before having read all input records. The second kind of approximation (representation-based) works on data-level, i.e., a certain degree of fuzziness is introduced such that for example one record might represent another one that would actually be part of the result.

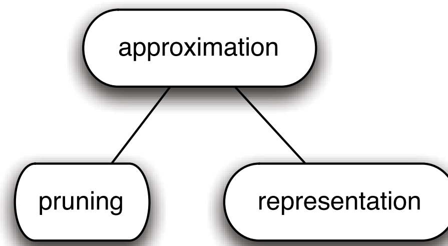


Figure 2.6: Classification of Approximation Techniques for Top-N and Skyline Queries

2.2.1 Pruning-Based Approximation

We have already discussed an algorithm [127] that applies pruning-based approximation in the context of skyline query computation in SSW overlay networks. Peers forward the query to peers that seem to provide relevant data and stop when the same peer has received a query twice. Thus, although there might be non-considered relevant data in the network, the algorithm stops.

Fagin [60] discusses several top-N algorithms for ranking multimedia data such as images and videos. In these systems data qualifications such as color are hard to deal with. For instance, the color is not simply defined as “red”. Instead, there is a degree of redness that ranges between 0 (not red at all) and 1 (totally red). In response to a query asking for red objects, a multimedia system might typically assign such a redness score to each object and output only the top N records. The result of such a query is a “graded” (or “fuzzy”) set [212]. The algorithms discussed by Fagin [60] are designed for middleware systems that have access to several data sources. For each data record we have m attributes and m sorted lists of records containing one entry for each record ranked by the corresponding attribute values. In this context, the usage of the threshold algorithm [59, 62, 81, 147] that we have already mentioned in Section 2.1.1 is proposed.

It begins with reading the sorted lists in parallel (sorted access). For each record that has been read in any list all "missing" attribute values are retrieved by random access. The algorithm stops reading the lists in sorted access when N objects with an overall grade at least equal to τ have been seen. τ is computed using the last seen attribute values retrieved by sorted access on the lists, i.e., the worst values seen so far by sorted access. Based on this threshold, Fagin presents an approximate variant of the threshold algorithm: θ-approximation. The θ-approximation to the top N answers (for ranking function r over database D) is a collection of N objects (and their grades) such that for each y among these N objects and each z not among these N objects: $\theta \cdot t(y) \geq r(z)$, $\theta > 1$. Only the stop rule of the threshold algorithm has to be adapted: as soon as at least N objects were seen whose grades are at least equal to τ/θ, the algorithm halts.

Another work [190] based on the threshold algorithm, an idea later on picked up by the KLEE framework [136], introduces probabilities into query processing and presents a family of approximate top-N algorithms that probabilistically try to predict when it is safe to drop candidate objects and to halt the scans on the sorted lists. What these algorithms do is similar to the θ-approximation technique. They approximate the result set by stopping the index scans as soon as a certain level of correctness is reached. Thus, they obtain a result set that still contains a certain level of approximation in comparison to the correct answer, which would be computed by the basic threshold algorithm.

Koltun et al. [114] present another notion of approximation for skyline queries that we also classify as pruning-based. Given a skyline query, the goal is to reduce the number of result records by computing a set of approximately dominating representatives (ADRs). The degree of approximation – denoted as ε – is defined by the user. A set of ε-approximate dominating representatives (ε-ADR) is defined as a subset A of D (D is the set of all records) that has the following property: For each $p \in D$ there is a representative $q \in A$ such that $(1 + \varepsilon) \cdot q$ dominates p, i.e., a skyline record p is represented by a record q, if the distance between p and q is smaller than ε. Koltun et al. provide algorithms which take a set of records as input and output the set of appropriate ADRs. For efficiency reasons the usage of a traditional algorithm is proposed to pre-process the input set. The resulting skyline is then provided as input for the algorithm to find the minimum set of ε-ADRs. Koltun et al. [114] proof that the problem of minimizing the number of ADRs for two dimensional data sets can be solved in polynomial time whereas the three or more dimensional case is NP-hard but has a polynomial-time logarithmic approximation. Thus, a polynomial time algorithm for the two-dimensional case is proposed. To calculate ADRs with three and more dimensions, Koltun et al. propose a greedy algorithm which approximates ADRs in any dimension. This is what Koltun et al. denote as a second kind of approximation. The result is no longer the optimum set of ADRs but an approximation of this optimum.

2.2.2 Representation-Based Approximation

We have already sketched an algorithm of this class in the context of skyline computation. Papadias et al. [153] propose an extension of BBS that based on the R-tree index computes hypothetical points representing all records represented by a region.

Kießling [111] examines techniques to enhance the computation and specification of preference queries in general – including Pareto (i.e., skyline) computation. Preference

queries allow the user to define soft constraints (i.e., preferences) instead of only hard constraints (i.e., standard selection predicates) within a query. Kießling presents a variety of preference constructors that allow for specifying user preferences. He distinguishes between base constructors and complex constructors that are applicable to skyline queries. As a novel semantic concept for complex preferences Kießling introduces the notion of "substitutable values" (SV-semantics), characterizing equally good values amongst indifferent values.

The key concept is to identify indifferent values, e.g., by using so-called d-parameters grouping ranges of values. In this way, a categorical view on numerical values is achieved so that certain indifferent values can be interpreted as "substitutable" or "equally good". Kießling shows that using such a categorical view may help to reduce the result size of skylines without sacrificing the importance of other user preferences defined in the query.

However, Kießling's focus is to personalize database queries by means of a semantically rich and flexible preference model for query composition and arbitrary databases. Thus, Kießling extends the constructor-driven foundation of preferences as strict partial orders in two ways: (i) by introducing d-parameters to model a categorical view on numerical data for base constructors, (ii) by enriching preferences by SV-semantics – preserving the strict partial order property even for Pareto preferences, prioritization, and numerical ranking.

We classified this kind of approximation as representation-based because the "meaning" of the data is changed by the introduction of SV-semantics, i.e., by applying this semantics the "value" of an attribute is set equal to another one.

2.2.3 Applicability to PDMSs

Intuitively, approximation offers the opportunity to reduce query execution costs not only in centralized systems but also in distributed environments such as PDMSs. However, most of the pruning-based techniques we have discussed are based on the threshold algorithm and rely on a special network architecture, which is not compatible to PDMSs. Another approach, which we classified as pruning-based, aims at computing a relaxed answer to a skyline query while relying on the set of all records or at least the non-relaxed answer to the skyline query as input. This requirement makes it unacceptable for PDMSs as these systems cannot provide that kind of global knowledge. Furthermore, we want to apply approximation in order to reduce execution costs. If we have to provide the answer to the skyline query as input, we obviously cannot achieve this goal.

On the other hand, the application of representation-based approximation techniques is much more promising. We have reviewed the application of SV-semantics as a representative of this class. If the user provides such information, this approach can also be used in PDMSs. Remember that the focus of SV-semantics is to personalize database queries using preferences whereas our goal is to reduce query execution costs. Thus, in this dissertation we focus on another representation-based approximation technique that exploits the information provided by the routing indexes to determine which records can be represented by others. Although we only consider this kind of representation-based approximation, SV-semantics could still be added to our algorithms.

2.3 Query Routing and Routing Indexes

Queries in distributed environments have to be rewritten into a set of subqueries that can be sent to neighboring peers in order to retrieve non-local data. One of the problems to be solved in this context is query routing. Whereas in structured P2P systems routing can efficiently be done by exploiting the rule according to which the data has been distributed among the peers in the first place, there is no such global knowledge in unstructured P2P systems which we could use to route queries efficiently to peers providing relevant data.

With respect to routing in unstructured P2P systems we distinguish between two main approaches (Figure 2.7): (i) flooding and (ii) semantic routing. Gnutella [70] is an example of the former kind of systems, i.e., each peer forwards the query to all its neighbors. They in turn proceed alike until a peer is found that can answer the query – resulting in flooding the network. Obviously, this results in a heavy load for the network. Thus, other systems use indexes that provide information about which peers hold relevant data to a query. Napster [144], for example, uses a centralized index, which makes Napster a somewhat hybrid approach since it uses a central component to route queries.

However, appropriate routing algorithms for use in PDMSs should work in a completely decentralized manner. Thus, each peer that receives a query has to decide which neighbors the query should be forwarded to, with the only help of the information received along with the query and the information stored locally. The application of routing indexes, which are very popular in unstructured P2P systems, is a possible solution to this problem. Thus, on the following pages we discuss related work on routing indexes and review several instances of the classes illustrated in Figure 2.7.

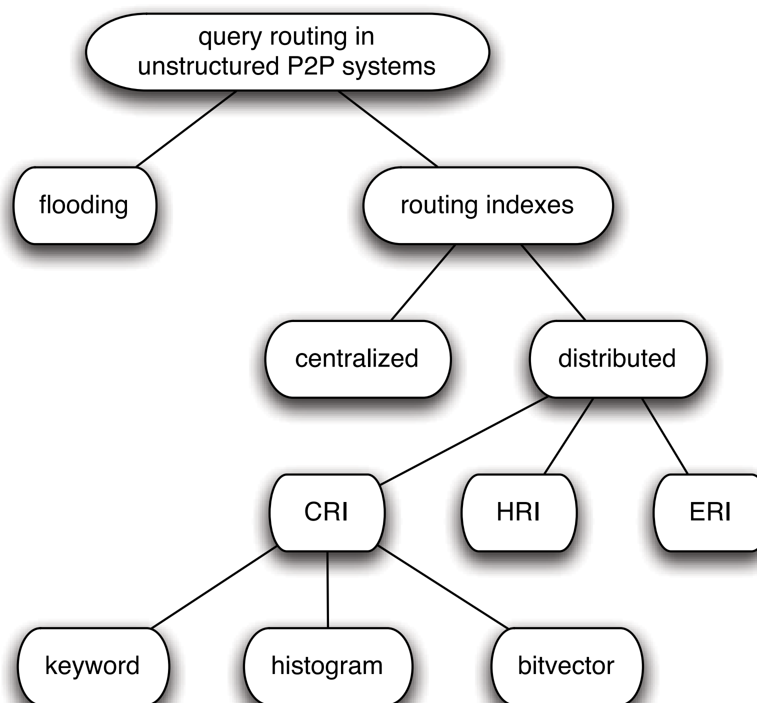


Figure 2.7: Classification of Routing in Unstructured P2P Systems

2.3.1 Routing Indexes

Crespo et al. [41] introduced the notion of routing indexes in unstructured P2P systems:

> The objective of a Routing Index (RI) is to allow a node to select the "best" neighbors to send a query to. An RI is a structure (and associated algorithms) that, given a query, returns a list of neighbors, ranked according to their goodness for the query. The notion of goodness may vary but in general it should reflect the number of documents in "nearby" nodes.

Thus, routing indexes are defined according to Definition 2.3.1.

Definition 2.3.1 (Routing Index (Crespo, Garcia-Molina))**.** *A* routing index *is a data summarizing structure that captures information about the data accessible via all neighboring peers. Each peer maintains exactly one such structure.*

In their work Crespo et al. consider a scenario in which documents are indexed according to a list of keywords or topics. Hence, documents are organized in categories and the index is a hash table of such categories. Several hash values may hash to the same bucket so that the count in a bucket represents the aggregate number of documents in those categories. Each peer maintains a local index that describes the locally stored documents according to the list of keywords. A routing index describes the data accessible via a peer's neighbors, i.e., RIs are used to index routes rather than destinations. Consequently, routing indexes in general provide information about neighbors but not for each peer in the system. More precisely, Crespo et al. distinguish between three types of routing indexes:

- Compound Routing Indexes (CRIs): CRIs contain (i) the number of documents available via each neighbor and (ii) the number of documents on each topic of interest (for each neighbor).

- Hop count Routing Indexes (HRIs): RIs of this type maintain one aggregation for each neighbor and each "hop" up to a maximum number of hops. Consequently, there are multiple aggregations per neighbor. For example, the 1 hop aggregation describes what data is reachable via each neighbor in a distance of 1 hop (i.e., the data stored locally at the neighbors). The 2-hops-aggregations for each neighbor subsume the local data of all peers reachable within 2 hops, the 3-hops-aggregations subsume the data of all peers reachable within 3 hops, and so on.

- Exponential Routing Indexes (ERIs): ERIs hold information about all accessible nodes in the system in only one aggregation per neighbor. The higher the distance of indexed data in terms of hops, the less influence they have on the aggregated value.

Queries are defined as conjunctions of keywords (indexed by the RIs). When a query is issued at a peer, the peer first processes the query locally. Then, it ranks its neighbors according to their goodness values (computed using its RI) with respect to the given query. Assuming that a stop value (a maximum number of result records) is given, the peer forwards the query to the first peer in the ranked list. Once the answer is received,

the peer checks if the stop condition is already fulfilled. If not, the query is forwarded to the next peer in the list. This goes on until finally either all neighbors have been asked or the stop condition is fulfilled. In addition to this sequential processing strategy, Crespo et al. also propose sending the query to several neighbors in parallel. However, they do not elaborate on this aspect and use the sequential strategy only.

Creation Routing indexes as proposed by Crespo et al. [41] are created and extended when peers join the network. The two peers that establish a connection between each other exchange information about the data accessible via them. This information is the aggregation of a peer's local index and the RI entries describing the data of its "old" neighbors. Both peers add a new entry for the new neighbor to their routing indexes with the received aggregated information. Afterwards, the two peers have to inform their old neighbors about the new data that is now accessible via them. Thus, they propagate the aggregated information to all such peers. These in turn proceed alike so that finally all peers have updated their routing indexes.

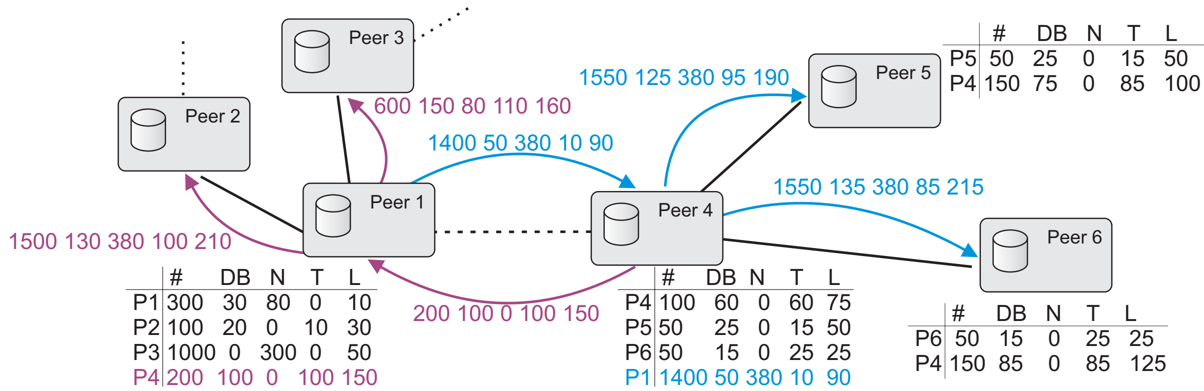


Figure 2.8: Example for Creating Routing Indexes in a Tree-Structured Network [41]. *Categories: databases (DB), networks (N), theory (T), and languages (L)*

Figure 2.8 shows an example of the construction process. P_1 and P_4 establish a connection that did not exist before (dashed line). Thus, they exchange the aggregation of their local indexes and their old routing indexes. Then, P_1 adds a new entry to its routing index describing the data it can access via P_4. P_4 does the same with the information received from P_1. Afterwards, both P_1 and P_4 propagate the information to all their old neighbors. The recipients update their RIs by simply replacing the old entries with the received information. For example, the message that P_1 sends to P_2 contains P_1's local index and the routing index information about data at P_3 and P_4. After all peers have propagated such information through the network, all routing indexes of all peers in the system are up-to-date.

Maintenance Whenever there is a change in the local data of a peer, it needs to propagate this change to all its neighbors such that they may update their routing indexes. In principle, the algorithm is the same as for the creation of RIs: (i) a peer recomputes its local index, (ii) aggregates its new local index and its RI information, and (iii) sends the resulting aggregation to all its neighbors – of course, for the aggregation the peer

needs to pay attention not to include the RI entry corresponding to the receiver of the update message. Then, the receiver updates its local routing index (replacing the old entry), computes new aggregates, and forwards them to its neighbors, which proceed alike. However, it is obvious that although this represents a basic solution to the problem, it does not scale well with the number of participating peers and the number of updates.

Cycles When the network contains cycles, the algorithms proposed so far might result in situations where an update runs in an infinite loop. Crespo et al. [41] discuss three strategies to solve this problem:

- *no-op solution*: the problem is ignored and we assume that cycles never occur,
- *cycle avoidance solution*: application of connection protocols to prevent cycles,
- *cycle detection and recovery solution*: cycles are allowed but actions are taken to recover from their effects. A unique identifier assigned to messages can be used to detect cycles. If a cycle is detected, a recovery procedure is started, e.g., only the first message is processed and all others are ignored.

Obviously, the third option provides the most flexibility and should therefore be chosen whenever possible. The approach presented by Crespo et al. [41] was only the beginning of RI usage for query routing in unstructured P2P systems. The concept has been extended and enhanced in other works. We are discussing some of them in the following.

2.3.2 Histograms and Bloom Filters as Routing Indexes

One of the works extending the basic routing index approach indexes structured data [158], i.e., a numerical attribute of data records is indexed instead of files. For this purpose, no longer keywords are used but one-dimensional histograms so that the distribution of numerical attribute values is captured. Each peer maintains one routing index (a histogram defined on the same numerical attribute) for each of its neighbors. A routing index describes all data records (with respect to the indexed attribute) accessible via the neighbor within a horizon of a predefined number of hops. Because of this definition, we classify this kind of routing indexes as a subclass of CRIs (Figure 2.7). Nevertheless, note that this still is a slightly different definition of routing indexes, since now there is one routing index per neighbor whereas in the original definition (Definition 2.3.1) there was only one routing index for all neighbors altogether. Definition 2.3.2 formalizes this new definition of routing indexes.

Definition 2.3.2 (Routing Index (Petrakis, Koloniari, Pitoura))**.** *A* routing index *is a data summarizing structure that captures information about the data accessible via one neighboring peer. Each peer maintains exactly one such structure for each of its neighbors.*

In addition to the routing problem, Petrakis et al. [158] also propose a technique to build network structures in which peers with similar data are clustered, i.e., there are many links to peers with similar data (short range links) and only a small number of links that connect peers of different clusters (long range links). When a peer P joins the network, the histograms describing the peers' local data (local indexes) are used to determine the difference between P's local data and the local data of other peers in the network. Based on the resulting similarity short and long range links are established.

With respect to routing range queries Petrakis et al. propose a sequential strategy where a query is routed to only one neighbor at a time – disregarding the possibility of exploiting parallelism. Each peer P that receives (or poses) a query at first processes it locally based on its local data. Then, the query is forwarded to the neighbor (out of the set of all neighbors) that promises to contribute the most results. When the answer is retrieved (answers are sent back the same way the query has been propagated on), P checks whether the desired number of matching data records (results) has been retrieved. If not, the query is forwarded to the second best neighbor of P (the third best, fourth best,...) until the desired number of records has been received.

In addition to the number of result records, there is another constraint for forwarding a query. A query is only forwarded (i) if the desired number of matching data records (results) has not yet been obtained and (ii) if the maximum number of visited peers has not yet been reached. If any of these two conditions is not fulfilled, the query is not forwarded. Although Petrakis et al. do not mention how they check these constraints, we assume they attach a simple counter of visited peers to the query message. Since query results are sent back to the initiator on the same route that the query has been propagated on and neighbors are queried sequentially, counting the current result size and the number of visited peers is straightforward. Note that these checks are only possible if parallelism is not practiced (depth-first traversal). However, parallelism would be a means to reduce processing time. To avoid cycles, each peer remembers the unique identifier assigned to each query and the peer the query has been received from.

If no short range neighbor with relevant data is found, the query is forwarded using a long range link hoping that even though the local routing indexes do not know of any peer with matching result records in the predefined horizon, then maybe there is a matching peer in the horizon of the long range peer.

Another approach [159] by Petrakis et al. applies similar strategies to routing indexes based on Bloom Filters [15]. In this work only queries based on keyword search are supported. Bloom Filters are bit vectors supporting membership queries used for probabilistic representation of sets. Such bit vectors do not provide any information about result cardinality, i.e., about the number of records that fulfill the condition defined by the query. The only question the applied Bloom Filters may answer is: "Does a keyword a occur in a set of documents?" As a consequence, a peer with routing indexes for its neighbors (indexing a set of predefined keywords) cannot determine how many result records are actually provided by them.

A common disadvantage of both approaches mentioned above is that the query routing algorithm does not exploit parallelism, which could considerably speed-up query processing. Another weakness is that both approaches do not consider any information about attribute correlation. If, for example, a query asks for documents containing two keywords a and b, the approach introduced above can identify neighbors that provide "interesting" documents. But it cannot decide if the queried keywords are contained in the same document or if there are only documents containing either a or b.

2.3.3 Bit Vectors as Routing Indexes

Marzolla et al. [134] present an approach for routing indexes based on bit vectors, which can also be considered a subclass of CRIs (Figure 2.7). Marzolla et al. apply them on

a numeric attribute whose range they partition into disjoint intervals of equal lengths so that one bit of the bit vector represents one of these intervals. Thus, the bit vector corresponding to a data record contains a “1” at the position of the interval that contains the indexed attribute value of the record. Aggregating routing indexes (as it is necessary to create them) is simply done by a bitwise “`or`” operation.

In order to support query routing, the queried attribute range has to be represented as a bit vector itself. By performing a bitwise “`and`” operation over the bit vectors of the routing indexes, relevant neighbors can easily be determined. Just like the approaches mentioned above, answers are sent along the path the query has been propagated on. To avoid cycles, the approach uses a spanning tree algorithm.

Marzolla et al. consider the problem of updates and insertions and propose an update propagation algorithm: whenever an update occurs, a new bit vector is created and compared to the old one. If these two vectors are different, the new vector is propagated to all neighbors. They in turn proceed alike and stop propagation when the received bit vector does not result in any changes of the indexes propagated to neighboring peers. In the worst case, all peers in the system have to update their routing indexes. Insertions and deletions can be treated as updates using the same algorithm. This kind of maintenance is almost the same as proposed by Crespo et al. [41] and likewise inefficient because a single update results in flooding the network.

Assume we have a network with many data records. Then, it is likely that a peer has data records with attribute values in many or even all of the bit vector intervals. It is even more likely that by aggregating the bit vectors of several peers, most if not all bits are set to “1”. Such routing indexes are of little use since queries are actually processed by flooding the network. Unfortunately, the evaluation presented by Marzolla et al. [134] is restricted to networks with very little data. Each peer holds at most one data record and approximately half of the peers do not provide any local data at all. Although Marzolla et al. [134] do not show the impact of having more data in the network, one can easily imagine that in this case query processing would result in almost flooding the network.

Just like the other approaches, this one does not consider queries defined on more than just one single attribute. However, a query on structured data is likely to be defined on several attributes. As mentioned above, without capturing attribute correlations many peers might be queried that cannot contribute to the result. Thus, the efficiency of query processing is limited because of inefficient query routing.

2.3.4 Routing Indexes Based on Indexing Clusters

An approach [53] very recently published by Doulkeridis et al., which also has to be considered a subclass of CRIs with respect to the classification of Figure 2.7, applies routing indexes based on the iDistance measure [100, 209] in conjunction with equi-width histograms to enable efficient routing of range and k-NN queries (k Nearest Neighbor) in super peer P2P networks. iDistance is an indexing method for similarity search that partitions the data space into n clusters. Each cluster C_i is defined by a cluster center K_i and a cluster radius r_i. Each data record p contained in a cluster C_i is assigned a one-dimensional iDistance value according to its distance to C_i's cluster center K_i. The data records themselves are stored into a B^+-tree using the iDistance values as keys. Applying this concept, similarity search is turned into range query processing.

In a super peer network, each super peer has a set of peers exclusively connected to the super peer. The super peer has some special tasks with respect to query processing, i.e., each query issued at any peer is sent to the corresponding super peer and optimized there. Hence, only super peers maintain routing indexes.

The construction of routing indexes follows a flooding approach. At first, each peer clusters its local data and sends the cluster descriptions (cluster centroid K_i and radius r_i) to its super peer. The original data remains at the peer. The super peer holds this information in a B^+-tree applying the iDistance technique in order to determine later on if a particular peer provides relevant data to a query. However, the super peer applies the clustering algorithm once again on the cluster information received from all its assigned peers – obtaining a set of hyper clusters that describe the data of all records provided by all its connected peers.

The hyper clusters are broadcasted among the super peers in the network and are further summarized in order to reduce the amount of data that needs to be kept in main memory. The result is a set of routing clusters maintained at super peer level and used for query routing across the super peer network. This means the routing clusters represent the routing indexes.

However, in order to process k-NN queries efficiently, super peers need additional information to estimate the number of records contained in a cluster. For this purpose, each super peer maintains an equi-width histogram for each cluster that any of its peers provides – capturing the distances of records to the cluster center. Furthermore, super peers maintain two additional histograms per hyper cluster. The first histogram indicates the number of clusters (of the peers) that a range query intersects – capturing the distance distribution of the peer clusters within a hyper cluster. The second histogram maintains an estimate of the number of data objects that the hyper clusters contain based on the radius of the intersection between query and cluster – summarizing the information of the peer clusters. This second histogram is attached to a hyper cluster and broadcasted along with the hyper cluster in the routing index construction phase.

Although this might be an efficient approach in super peer architectures with only a small number of super peers, we argue that the amount of additional information (three-level clustering plus several kinds of histograms) the super peers need to maintain in main memory is relatively high. Furthermore, construction and maintenance of the routing indexes and histograms is based on flooding the super peer network whenever a change occurs – this might be a local data update of a peer or a peer joining or leaving the network. Especially in highly dynamic networks or networks with a large number of super peers this is no satisfactory solution.

2.3.5 Applicability to PDMSs

In this section we have discussed several examples of routing indexes. Some of them have been developed to index keywords in files, others to index individual attributes of structured data without capturing attribute correlations. However, as top-N and skyline queries are defined on numerical attributes, the ones most relevant to our work are based on histograms. As we have already discussed, having no information about attribute correlation but only information about each attribute in separate, leads to a high number of unnecessary false positive routing decisions, i.e., the query is forwarded to many peers

that cannot contribute to the result. Considering the fact that queries (especially skyline queries) are likely to be defined on multiple attributes, we argue that routing indexes capturing attribute correlations should be preferred. We also reviewed one approach which considers indexing multidimensional numerical data. However, although this might be an efficient approach in super peer architectures with only a small number of super peers, we argue that the amount of additional information (three-level clustering plus several kinds of histograms) the super peers need to maintain in main memory is too high. In principle, however, all the approaches to index numerical data could be adapted to work in PDMSs. To overcome the problem of schema heterogeneity, we could, for example, define routing indexes on attributes of a peer's local schema and index the data of neighbors that can be mapped to those attributes.

Still, there is another important aspect, even though it is ignored by most approaches: index construction and maintenance in the presence of data updates and/or changes in the network structure. Even the approaches that acknowledge this as a problem only propose a simple flooding strategy. Intuitively, it is obvious that this cannot be efficient in large or highly dynamic networks as flooding the network is likewise expensive for index maintenance as it is for query processing.

2.4 Histograms and Maintenance

In the previous section we have reviewed related work on routing indexes and identified those based on histograms (Figure 2.7) as being most relevant with respect to processing queries on structured numerical data. As histograms are a very popular approach to summarize numerical data and as there has been considerable research, let us first discuss some approaches proposed by the literature and afterwards review several techniques regarding their maintenance.

Assume we want to capture the value distribution of a particular numerical attribute. For this purpose, histograms use buckets defined by lower and upper boundaries describing the range of values they represent. In addition to these boundaries, a bucket is assigned statistical information, i.e., the number of records whose attribute values lie within that range. Some of the most popular histograms are: equi-width histograms, equi-depth histograms, and compressed histograms. For equi-width histograms the value range is partitioned into multiple buckets that all have equal extensions, i.e., the distances between upper and lower boundaries are equal. In contrast to equi-width histograms, the goal in construction of equi-depth histograms is to create buckets that represent equal numbers of records. Consequently, buckets might have different extensions/sizes. For compressed histograms, the k most frequent attribute values are stored into singleton buckets (each capturing not a range of attribute values but only one) whereas the others are partitioned according to the rules of equi-width or equi-depth histograms. There are many more variants of histograms and construction algorithms. For the sake of brevity, we refer to [99] for a comprehensive study.

Most histograms are only defined on one attribute value. Using multiple one-dimensional histograms to estimate the result cardinality for queries defined on multiple attributes is only possible by assuming independence between attributes. In reality, we usually observe correlation between attributes and the assumption of independence often

leads to bad approximations of result cardinalities [161]. This is disadvantageous not only for routing indexes but also for centralized systems and the problem of cardinality estimation. The first histogram developed to counteract this problem is the two-dimensional equi-depth histogram proposed by Muralikrishna et al. [142]. By defining histograms on multiple attributes, correlations can be captured and cardinalities can be estimated more precisely.

However, most multidimensional histograms [80, 142, 161] are static and need to be reconstructed each time the data they summarize is updated. This is expensive not only in centralized environments but also in distributed environments, for which reconstructing routing indexes is even more expensive. Thus, in the following we review some strategies designed for efficient incremental maintenance of histograms.

2.4.1 Maintenance

First approaches for incremental maintenance of one-dimensional equi-depth and compressed histograms were proposed by Gibbons et al. [69]. In addition to histograms, the system also maintains a backing sample, i.e., an up-to-date random sample of the records currently stored in a relation. In order to adapt histograms, an algorithm based on split and merge operations is proposed. Whenever an update occurs, the backing sample as well as the corresponding histogram bucket are adapted. If the count of a bucket, i.e., the number of records it represents, exceeds a predefined split threshold, the bucket is split up into two. The boundaries of the resulting buckets are chosen according to the backing sample. If now the number of buckets exceeds the maximum number of buckets the histogram is allowed to use, other buckets need to be merged. Thus, two adjacent buckets whose combined count does not exceed the split threshold are merged. Likewise, if due to deletions the number of records within a bucket falls beyond a predefined merge threshold, the bucket is merged with one of its adjacent buckets and the bucket with the largest count is split up.

The concept of query feedback was first introduced by Chen et al. [38]. In this approach the data distribution is approximated by a curve-fitting function. In order to adapt this function, the approximating distribution is adjusted according to query feedback using a technique based on the recursive least-square-error. However, the concept of query feedback has been used later on also in the context of histograms. LEO (DB2's LEarning Optimizer) [179] exploits query results to validate database statistics and to compute adjustment factors in order to repair incorrect statistics and cardinality estimates of query execution plans. Application of feedback in LEO works in four steps: (i) retaining the query plan, (ii) monitoring query execution and collecting feedback information, (iii) analyzing feedback information and computing correction values, and (iv) considering correction values for query planning. In contrast to most other approaches, the histograms are not changed, rather both the original histogram and the correction values are considered in order to estimate result cardinality.

Updating multidimensional histograms using query feedback was considered by Aboulnaga et al. [3]. Because of the applied grid structure (defined by the buckets' boundaries) and the aspect of self tuning, the approach is often referred to as STGrid. Applying this approach, both the buckets' counts as well as their boundaries are adapted. Adjacent buckets of the initial histogram share bucket boundaries, i.e., there is no space in between

buckets, and the regions of all buckets altogether cover the entire data space defined by the indexed attributes. In order to update the bucket counts, the algorithm exploits feedback from issued queries. At first, the algorithm needs to find all buckets whose regions overlap the queried region because all these regions potentially represent data that was part of the query answer. Based on these regions and the overlap, the difference between the estimated number of result records and the retrieved number is determined. The bucket counts are adapted in proportion to their current count, i.e., assuming the higher a bucket's count the higher is its "blame" on the incorrect estimate. Bucket boundaries are adapted periodically based on the split/merge algorithm proposed by Gibbons et al. [69], which considers each attribute in isolation. In contrast to the original algorithm, more than two buckets can be merged and a bucket can be split up into more than two buckets. As STGrid does not hold any samples, the buckets resulting from a split operation have equal sizes. In order to ensure that the maximum number of buckets is not exceeded, some consecutive buckets need to be merged. Hence, if the difference between consecutive buckets is less than a predefined percentage of the number of indexed records, they are merged. The disadvantages of this algorithm are due to the relatively rigid grid structure. Thus, a split operation with respect to one dimension does not affect only one bucket but, because of the multidimensionality, all buckets within a slice of the data space. Although the split decision might be worthwhile for some of the buckets, it is suboptimal for others.

For all histograms discussed so far buckets are not allowed to overlap. This is different for GENHIST [80], which allows buckets to have variable sizes and to overlap. During construction of these histograms, the grid structure (created by the buckets partitioning data space) is redefined in each step and those cells with the highest densities are made to buckets. However, this approach does not consider the problem of maintenance and the configuration of its many parameters is complex. Another approach that allows buckets to "overlap" but considers updates is STHoles [22]. STHoles has a hierarchical structure, in which buckets might contain others, i.e., they might contain holes represented by child buckets. Just like STGrid, STHoles uses query feedback to update histograms. The distribution of buckets within the data space in STHoles depends on the query workload, i.e., the more frequently a region is queried, the more buckets are used to describe the data distribution within that region. The histogram is initialized as one large bucket that covers the whole data space. Then, in dependence on the queried region a new bucket is created, i.e., a hole is drilled into an already existing bucket. In order to limit the number of buckets, buckets with similar frequency distributions are merged. Although this approach eliminates several problems of STGrid, some problems remain, e.g., it might take a long time to correctly represent larger changes in the data distribution due to updates. Furthermore, as a general problem of all approaches using query feedback for index maintenance, estimates for regions queried only rarely are likely to be bad.

Srivastava et al. [178] presents ISOMER (Improved Statistics and Optimization by Maximum-Entropy Refinement), an algorithm for feedback-driven histogram construction. For this purpose, the algorithm collects query feedback records (QFRs) consisting of predicates and their selectivities. The algorithm uses the maximum-entropy principle to approximate the true data distribution by the "simplest" distribution that is consistent with the set of all currently valid QFRs. ISOMER uses STHoles as base structure but uses a slightly different algorithm for creating and merging buckets. QFRs are collected

and used in batches to refine the histogram, e.g., during periods of light load. When updating an existing histogram, all old QFRs that conflict with the new ones need to be identified and removed. Afterwards, new buckets (holes) are created so that the current set of QFRs is represented by the histogram. Finally, buckets are merged in a similar fashion as proposed by Bruno et al. [22].

2.4.2 Applicability to PDMSs

In this section we have reviewed maintenance strategies for histograms with the intention of finding appropriate strategies that we can use to build and maintain routing indexes in PDMSs. The main problem with all these approaches is that they have been designed to update histograms maintained for estimating selectivity or result cardinality in centralized systems. Thus, the split and merge strategies used to construct and update these histograms aim at producing buckets that capture areas with similar frequency distributions. This often results in rather large regions represented by a single bucket. Furthermore, the approaches discussed so far assume that the buckets cover the whole data space. STHoles makes even use of a bucket (the root bucket) that covers the whole data space.

When estimating selectivity in centralized systems, as the strategies have been developed for, the worst thing that could happen is choosing a suboptimal query plan while the correct result would still be retrieved. For routing indexes query execution costs could be unnecessarily high (false positive routing decisions) or query results could be incomplete (false negative routing decisions). If the histogram of a routing index has large buckets (no matter how many data records they represent), the routing algorithm most likely always forwards the query to all its neighbors. This is true not only for range queries but also for skyline and top-N queries. Although the result cardinality might be estimated correctly, the goal of applying routing indexes, which is the reduction of network load and the number of involved peers, is not achieved. Hence, it must be the primary goal of a routing index to indicate what areas do *not* provide any data at all such that the peer can safely prune neighbors from consideration. Then, only the secondary goal is to form buckets capturing uniform data distributions.

Another disadvantage of the approaches based on query feedback reviewed in this section is that histograms are only updated in subspaces which are queried frequently. Applying such a strategy for routing indexes would mean that when a peer updates its local data such that it afterwards provides data with attribute values completely different from the ones it has provided before, then it will not receive any queries from its neighbors referring to its new data. Thus, relevant data that should be considered is missed.

2.5 Distributed Query Processing Strategies

As already mentioned above, in structured (DHT-based) P2P networks the same rule that was used to distribute data among peers can be used to efficiently route queries to only those peers that contribute to the result. As we assume unstructured P2P networks to underly PDMSs, redistributing or replicating data is not possible. In contrast, peers have to make routing decisions with the only help of schema mappings and routing indexes.

Still, it is possible to consider a super peer architecture as basis in which only the super peers as a backbone network participate in a PDMS. Then, super peers make the data of their subordinate peers accessible to other super peers in the network. Although the application of routing indexes is recommendable and promising for every query processing strategy, they can be considered optional. Thus, in the following we discuss techniques for query processing that are independent from the application of routing indexes.

2.5.1 Data Shipping and Query Shipping

In P2P systems there are two basic paradigms for processing queries that can also be applied to PDMSs: data shipping (DS) and query shipping (QS) [115]. Applying data shipping, all data identified as being relevant to a query is transferred (shipped) to the initiator, which processes the answer to the query locally based on its local data and the data received from other peers in the network. When applying query shipping, the initiator sends the query to other peers in the system. In contrast to the data shipping approach, parts of the query are already processed at the peers that provide relevant data. Only the data they cannot process any further is sent to the initiator, which then computes the final result based on its local data and the preprocessed data received from its neighbors. It is obvious that such preprocessing reduces the amount of transferred data as well as network load and execution costs. Both basic paradigms, data shipping and query shipping, exploit the fact that peers can work in parallel on the same query.

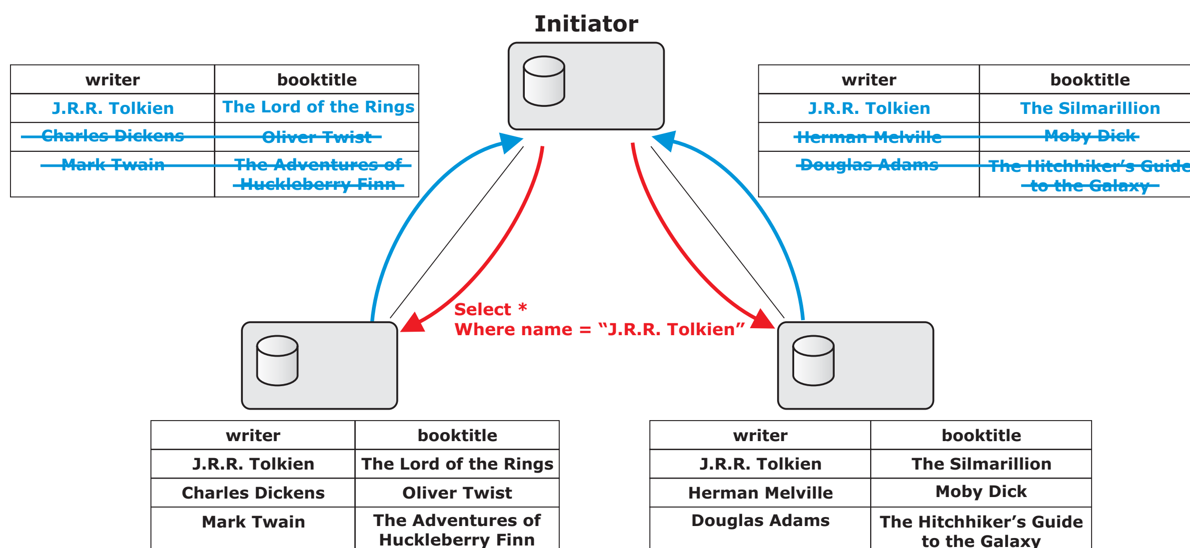


Figure 2.9: Query Shipping vs. Data Shipping

Figure 2.9 illustrates the differences between data shipping and query shipping. Let us assume the initiator forwards the query to two neighboring peers. Applying data shipping, both send much more data to the initiator than necessary, namely all their local data, since they do not evaluate the query locally. However, when applying query shipping, the two peers that receive the query evaluate the predicate "name = J.R.R. Tolkien" on their local data and thus send much less data to the initiator.

2.5.2 Mutant Query Plans

An interesting approach for query processing in distributed environments on XML data is the application of Mutant Query Plans (MQPs) [154]. The principle is illustrated in Figure 2.10. and basically combines data shipping and query shipping. An MQP is a query plan graph that in addition to URLs (Unified Resource Locators) may contain URNs (Unified Resource Names). While the former refer to peers in the network, the latter refer to abstract resource names. A peer receiving an MQP (serialized for transmission in XML) has a local catalog that maps URNs to URLs. Using this catalog the peer is able to replace URNs with URLs of peers holding the data referred to by the URNs. As the information provided by the catalog may be incomplete, a peer is usually not able to replace all URNs of a query. Thus, the query is forwarded to another peer that hopefully will be able to do so. In addition to replacing URNs the peer itself tries to compute at least parts of the query locally based on its local data. The resulting XML fragments are inserted into the MQP. Then, the mutated query plan is sent to a peer that knows how to resolve at least one of the remaining resources. By chaining the query in this manner through several peers, the answer to the original query can be processed in a distributed manner – sharing load among multiple peers.

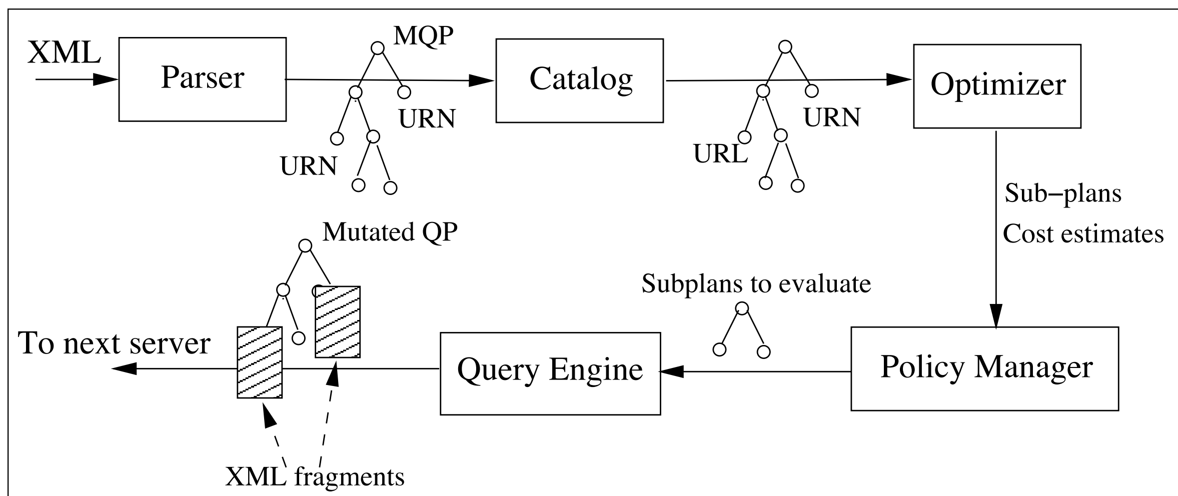


Figure 2.10: Mutant Query Plans [154]

A formal model for MQPs is presented by Abiteboul et al. [2]. The key contribution is to provide an algebraic model and thus a language that enables formulating distributed computations on XML data. For this purpose, XML documents contain service calls that specify the receiving peer, the name of the service as well as a set of parameters.

2.5.3 Incremental Message Shipping

So far we have only distinguished strategies by what data is sent through the network and whether parts of the query are evaluated at remote peers. However, usually peers receiving a query forward the query again such that chains with lengths of several hops occur. Let us assume each peer in such a chain always waits for the answers of its neighbors before sending its own answer. This works well in static systems but leads

to great difficulties in dynamic environments with peer failures. To avoid having peers wait for an answer of a crashed peer for too long, timeouts can be used. When such a timeout elapses, the peer processes its answer only based on the already received answers. Nevertheless, the user might still wait a long time until he/she is presented an answer. We have proposed an incremental strategy (Incremental Message Shipping, IMS) [107] to overcome these shortcomings. Peers forward result records as soon as they are known. Thus, a queried peer is likely to send more than just one answer message. Consequently, this results in a higher number of messages and thus a higher network load but has the advantage that even when peers crash, answers arrive at the initiator and first results are output to the user at an early stage.

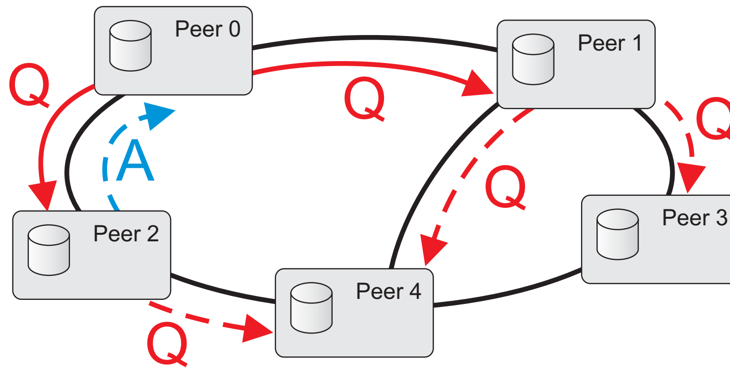


Figure 2.11: Incremental Message Shipping - Query Propagation

Figure 2.11 shows an example of IMS – illustrating the first phase of query propagation. Let us assume P_0 issues a query and sends it to its neighbors (P_1 and P_2). P_1 again forwards the query to P_3 and P_4. P_2 also forwards the query to P_4 not being aware that the query has already been sent to P_4 by P_1. P_2 already sends an answer message (evaluated on its local data) to P_0. P_1 does not send such a message because it has no local data records matching the query. In the next step, P_0 can already output an inter-

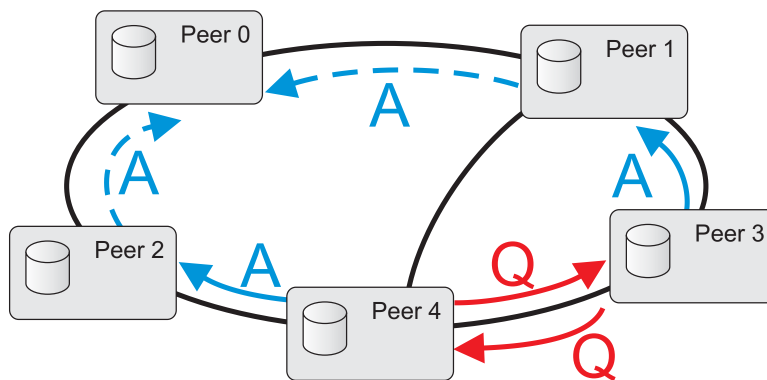


Figure 2.12: Incremental Messsage Shipping - Answer Propagation

mediate result based on the local data of P_0, P_1, and P_2 to the user. Figure 2.12 depicts the following steps in processing the query. P_3 processes the query received from P_1, it forwards the query to P_4, and sends an answer to P_1. Likewise, P_4 processes the received

messages. Assume the message from P_2 has been received first. Then, P_4 processes this query, forwards it to P_3, and sends an answer to P_2. The message received from P_1 is neglected because the answer to the query with respect to P_4's local data is already sent to the initiator via P_2. The application of a unique query identifier (e.g., based on the peer ID of the initiator and a timestamp) enables P_4 to detect such a cycle. Both queries that have been forwarded in the last step to P_3 and P_4 can safely be neglected due to the same reasons. P_1 and P_2 forward the answers to P_0, which now can compute the final result and output it to the user.

The difference to QS is that peers can send multiple answer messages or none at all. Hence, in general more messages are sent using IMS. However, in networks with only a few peers contributing to the result, IMS may perform better than QS even though the network might be rather static. The reason is that QS demands each peer to send one answer message in any case so that the sender of the query may stop waiting. As this is not necessary using IMS, the number of messages might actually be lower in comparison to QS when only a few peers contribute to the result.

2.5.4 Applicability to PDMSs

With respect to PDMSs we cannot apply MQPs in their original sense. The main reason is that with the absence of global knowledge we cannot create a global query plan at the initiator. In addition, peers in a PDMS cannot contact arbitrary peers since a mapping between two such peers needs to exist. Furthermore, answers to a query *must* be routed back the same way as the query because not only does the query itself need to be rewritten but also the data records of an answer such that finally the initiator receives a set of records in its local schema. Another weakness of MQPs is that they do not exploit parallelism, which should be considered in distributed environments in order to reduce query response time. However, there are still some interesting aspects a strategy applicable to a PDMS should use. These are chaining and having non-initiator peers compute partial or intermediate results and attaching them to the query. QS and IMS are both applicable to PDMSs if the query is routed back the same way it has been propagated on. Furthermore, we need to add a query rewriting step before forwarding the query and a result transformation step before sending an answer.

2.6 Query Rewriting

A basic prerequisite for query rewriting is, of course, to define mappings between schemas. Hence, before going into details on query rewriting let us first discuss the basic approaches to define mappings between schemas. For data integration the two most common approaches to express mappings are: *local-as-view* (*LAV*) and *global-as-view* (*GAV*) [86, 119]. The basic idea of these approaches is to express a mapping between two schemas as views. For this reason, the definition of PDMSs (Definition 1.1.1), as given in the introduction of this dissertation, uses queries to express mappings/assertions. Such queries are defined on the relations of one schema and create relations of the other schema (without loss of generality referring to the relational data model). Assuming we have a mediator system [68, 124] with a global schema and several connected sources with

heterogeneous local schemas, the difference between LAV and GAV is which schema is described by the views:

- the global schema (GAV) or
- the local schemas (LAV).

This means that in order to define mappings, GAV approaches only use assertions of the form $q_{\mathcal{S}_\mathrm{P}} \subseteq g$ (with $q_{\mathcal{S}_\mathrm{P}}$ being a query referring to relations in the source schemas and $g \in \mathcal{G}_\mathrm{P}$ referring to a relation in the global schema) and LAV approaches only use assertions of the form $s \subseteq q_{\mathcal{G}_\mathrm{P}}$ (with $s \in \mathcal{S}_\mathrm{P}$ referring to a relation in the source schemas and $q_{\mathcal{G}_\mathrm{P}}$ being a query referring to relations in the global schema). The differences between GAV and LAV can best be illustrated with an example. Thus, let us consider a mediator system with relational data as an example.

Global-As-View (GAV)

For each global relation we need to define a view (query) over the sources' local schemas. Typical for such views is that they contain several union operations to combine the results of multiple subqueries. These subqueries describe how to extract tuples from the sources. Figure 2.13 shows an example with P_0 serving as the mediator – the schema of P_0 serves as global schema. The views maintained by P_0 describe how to integrate the data from P_1, P_2, and P_3 into the global schema.

When a query Q (formulated in the global schema) is issued at P_0, all global relations referred to by Q are replaced by their view definitions (unfolding). To execute Q, we first need to compute the referenced views and then all the other operations (select, project, join) being part of Q's definition.

Obviously, the main advantage of GAV is that processing the query means low algorithmic effort: we simply need to replace references to global relations with their view definitions and process the resulting query. On the other hand, some knowledge might get lost when modeling the global relations, e.g., the data of the global relations might already be stored in a joined format at a source. Furthermore, local constraints at the sources cannot be modeled at the global level. Another important disadvantage is that whenever a data source enters or leaves the network, the global relations and their view definitions have to be adapted in order to reflect the new situation.

Local-As-View (LAV)

In contrast to GAV, the local schema of each source peer in LAV is described as a view over the global schema. Figure 2.14 shows how such mappings could look like in the network of Figure 2.13.

The problem we encounter for processing a query formulated in the global schema is that we can no longer simply replace a part of the query with the view definitions. Instead, we have to *rewrite* the query in order to access the data of the sources. In general, the rewriting process produces a set of queries whose results are converted into the global schema and aggregated into the final query result. The literature proposes several algorithms dealing with this problem of "query answering using views" [84, 122]: the Bucket Algorithm [74, 123, 124], Inverse Rules [54, 55], and MiniCon [162, 163].

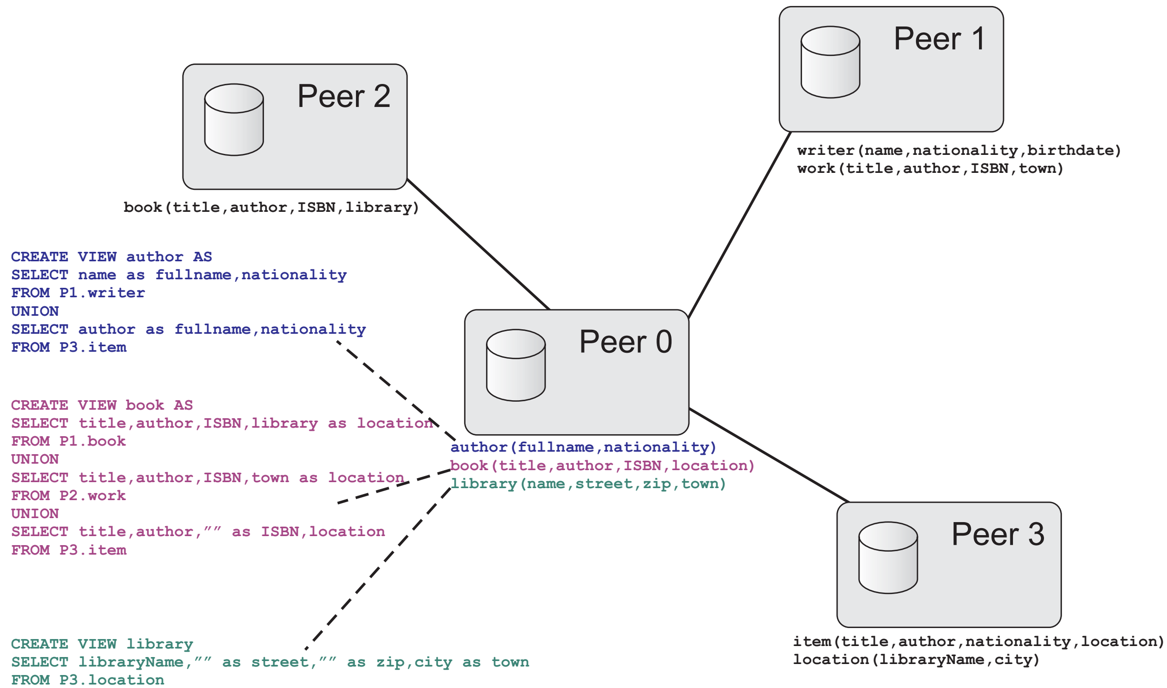


Figure 2.13: GAV Example with P_0 as Mediator

The advantage of LAV over GAV is that when a peer joins or leaves the network, the global schema as well as the mapping definitions of all the other sources remain unaffected. Furthermore, associations and constraints defined locally at the sources may be considered. As already indicated above, these advantages do not come for free: they require far more complex query processing techniques that include query rewriting.

Combinations of GAV and LAV – GLAV and Both-As-View

As we have already discussed above, both LAV and GAV have several disadvantages. Hence, approaches have been developed that combine LAV and GAV. One of them is GLAV (Global-Local-As-View) [66]. It promises to combine the expressive power of both LAV and GAV. For LAV a mapping to a source relation is defined by using a view formulated on the global schema. For GAV a mapping to a global relation is defined as a view formulated on source relations. Applying the GLAV approach, mappings are defined by using two views, i.e., a view on the local schema is defined as a view on the global schema – allowing recursive queries over the sources. Thus, when applying GLAV, the global schema and mappings do not have to be adapted each time a new source joins the network. Whereas for LAV-style mappings the global schema needs to represent all schema elements that are shared by multiple source relations (to allow joins between them), GLAV-style mappings can express joins between multiple source relations in the mapping definition so that these join attributes do not have to be part of the global schema. Furthermore, query processing can benefit from the GAV assertions.

The Both-As-View (BAV) approach [151] provides a framework of reversible schema transformation sequences (pathways). Using BAV it is possible to extract both a definition of the global schema as view over the source schemas and definitions of the

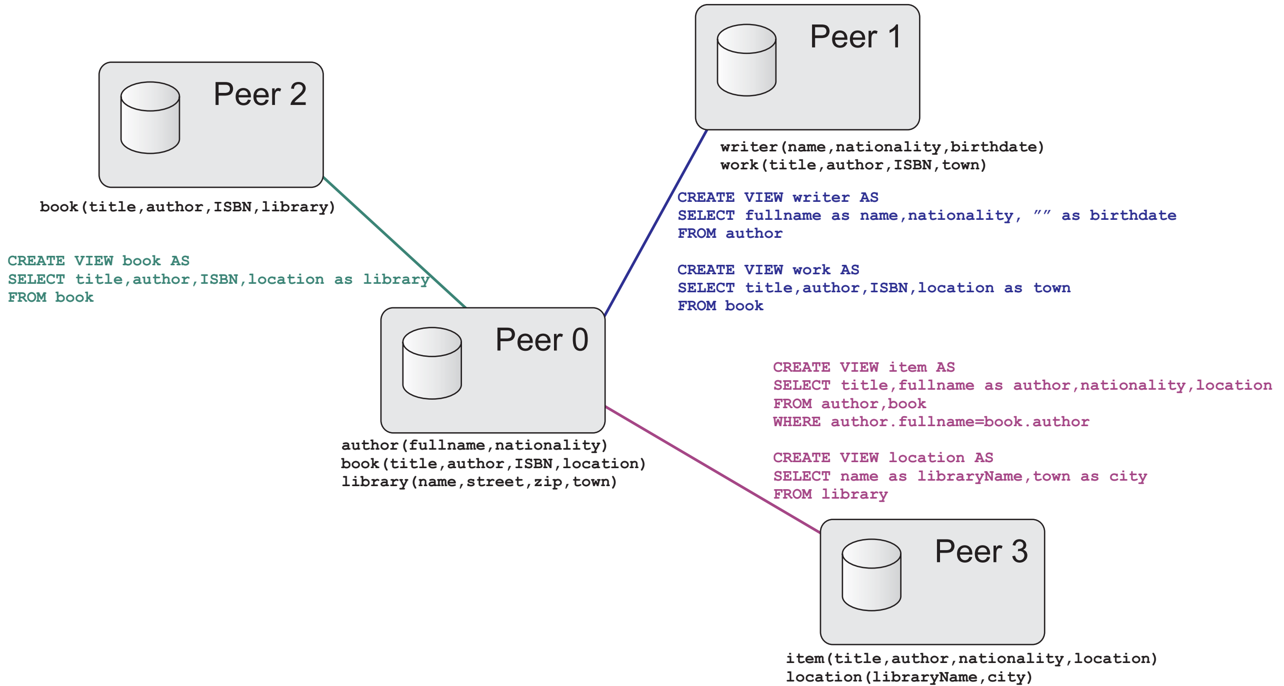


Figure 2.14: LAV Example with P_0 as Mediator

source schemas as views over the global schema. On the other way around, BAV pathways can be extracted from GAV and LAV view definitions. Furthermore, BAV supports the evolution of global and local schemas, allowing the transformation pathways and schemas to be incrementally modified. Thus, BAV combines the advantages of both GAV and LAV. However, the disadvantage is that pathways are often very fine granular so that query optimization might be expensive for the corresponding LAV and GAV view definitions.

Because of the advantages such as low-effort extensibility that LAV entails for dynamic environments, we favor LAV over GAV. Being a combination of both approaches, GLAV [66] would be another interesting choice we could use to define mappings in our system. However, as LAV is the more challenging part, we limit our considerations to LAV-style mappings.

2.6.1 Conjunctive Queries

As most related work on query rewriting focuses on conjunctive queries in the context of relational data, we introduce this kind of queries in the following and sketch how query rewriting based on them works. At first, we give a theoretical introduction to the topic and then illustrate the rewriting process with an example.

Conjunctive queries [193,194] represent select-project-join queries and do not consider any kind of grouping, sorting, aggregation, or any other kind of higher level operators. A conjunctive query can be written in the form [25]:

$$\{x | \exists\, y : body(x, y)\} \tag{2.1}$$

where $body(x, y)$ is a conjunction of atoms (function-free first-order logic (FOL) for-

mulas that may contain constants and whose relation symbols refer to relations in the given schema) involving the free variables $x = x_1, \ldots, x_n$ (also distinguished or exported variables), the non-distinguished variables $y = y_1, \ldots, y_m$ (also existentially quantified variables), and constants. Thus, the conditions given by the atoms in the body are connected via logical "and" operations. A query containing atoms connected via logical "or" operations can still be expressed by combining the results of multiple conjunctive queries [33], which corresponds to the application of a union operator in relational algebra. Futhermore, the conjunctive queries we consider in the following may contain built-in arithmetic comparison predicates using $\leq, <, \geq, >, =$, and $\neq$. Hereafter, they are referred to simply as predicates or conditions.

The datalog notation for conjunctive queries has been widely adopted. A conjunctive query without built-in comparison predicates in datalog notation is a logical rule adhering to the following form [84]:

$$q(\bar{X}) : -r_1(\bar{X}_1), \ldots, r_n(\bar{X}_n) \tag{2.2}$$

where q and $r_1, \ldots, r_n$ are predicate names referring to database relations. The atom $q(\bar{X})$ is called the *head* of the query and refers to the answer relation. The atoms $r_1(\bar{X}_1), \ldots, r_n(\bar{X}_n)$ are the *subgoals* in the body of the query. The records $\bar{X}, \bar{X}_1, \ldots, \bar{X}_n$ contain either variables or constants. For query rewriting we limit the expressiveness of datalog and require that the query is non-recursive and safe [193], i.e., $\bar{X} \subseteq \bar{X}_1 \cup \cdots \cup \bar{X}_n$, that is every variable appearing in the head must also appear in the body.

Datalog queries assign variables and constants to attributes of a relation using the position of a variable in the schema to determine the corresponding attribute. Variables may be used multiple times within the same datalog rule. Join predicates, for instance, are formulated by assigning the same variable to multiple attributes. Datalog queries may also contain subgoals whose predicates involve arithmetic comparisons $\Theta = \{\leq, <, \geq, >, =, \neq\}$. Thus, a query in general has the following form:

$$q(\bar{X}) : -r_1(\bar{X}_1), \ldots, r_n(\bar{X}_n), c_1, \ldots, c_m \tag{2.3}$$

where $c_1, \ldots, c_m$ are binary boolean logic expressions of the form $X\ \theta\ Y$ with $\theta \in \Theta$ referring to variables and constants. For all involved variables we require that if a variable X appears in a subgoal of a comparison predicate, then X must also appear in an ordinary subgoal [84].

Figure 2.15 illustrates the relationship between SQL, XQuery, and datalog. It shows an example SQL query on a relational schema and the same query formulated in XQuery referring to XML structures representing the same data. The name of a *context node* in the XML structure corresponds to the name of a relation in the relational data model. As both queries involve only select-project-join operations, both queries can be mapped to the same datalog query.

With respect to datalog queries we distinguish between:

- *exported symbols* or *exported variables* such as x and y in Figure 2.15,
- *literals* or *subgoals* such as $writer(x, y, n)$,
- *constants* (e.g., *Germany*), and
- *conditions* (e.g., $\{n =$"*Germany*"$\}$).

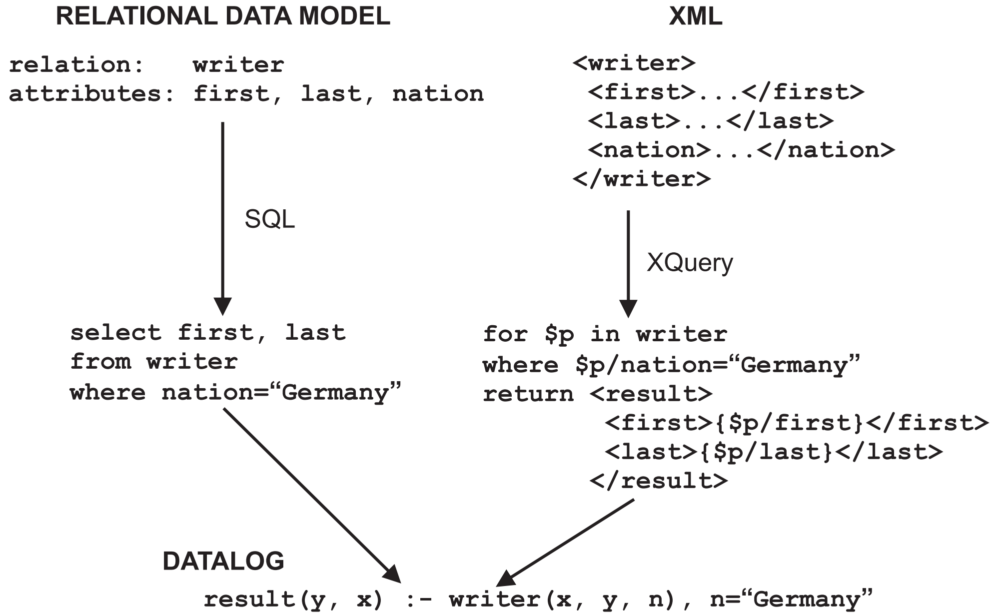


Figure 2.15: Conjunctive Queries on Relations and XML Data

According to this we denote:

- the set of symbols (variables) of a query Q as $\mathcal{S}(Q)$,
- the set of exported symbols as $\mathcal{E}(Q)$,
- the set of literals/subgoals as $\mathcal{L}(Q)$,
- the set of constants as $\mathcal{C}(Q)$, and
- the set of conditions/predicates as $\mathcal{P}(Q)$.

2.6.2 Query Containment and Containment Mapping

As this dissertation focuses on LAV-style mappings, we need to consider how these mappings can be used to rewrite a query. For this purpose, let us consider a single data integration system that might be part of a PDMS in the sense of Figure 1.4 and discuss the main characteristics of an appropriate rewriting.

The input to an algorithm for query rewriting using views is a query Q formulated on the global schema and a set of views $\mathcal{V} = \{V_1, \ldots, V_n\}$. The task of the algorithm is to find a rewriting Q' formulated on the views only. With respect to such rewritings we distinguish two categories: equivalent rewritings and contained rewritings. An equivalent query rewriting provides all answers to the query whereas a contained query rewriting possibly provides only a subset. Definition 2.6.1 formally defines the two notions [84].

Definition 2.6.1 (Query Containment and Equivalence)**.** *A query Q is said to be contained in a query Q', denoted by $Q \sqsubseteq Q'$, if for all database instances D the set of records computed for Q is a subset of those computed for Q', i.e., $Q(D) \subseteq Q'(D)$. The two queries are said to be equivalent if $Q \sqsubseteq Q'$ and $Q' \sqsubseteq Q$.*

In PDMSs or data integration systems in general we have to deal with a possibly large number of peers (sources) and thus a large number of views describing mappings between them. Although we do not assume that each peer maintains views to all other peers in the network, depending on the application and the network structure a peer might still have a high number of neighbors, i.e., peers that it has established mappings to. In contrast to query optimization in centralized systems, which use materialized views to accelerate query processing, the data provided by the sources in a PDMS might often be incomplete, i.e., a source might only store some of the records that satisfy its view definitions. In this context, Lenzerini et al. [26, 119] distinguish three cases: views can be *sound* (i.e., the source extension provides any subset of the records satisfying the view definition), *complete* (i.e., the source extension provides any superset of the records satisfying the view definition), or *exact* (i.e., the source extension provides exactly the set of records satisfying the view definition). Another distinction we need to make is whether the set of objects contained in the domain (world) is exactly the same as the set of objects represented by the views (closed-world assumption – CWA) or if there might exist any other objects besides those stored in the views (open-world assumption – OWA). As in a PDMS we can usually neither guarantee that a view is complete or exact nor that all existing objects are represented by the sources; in this dissertation we adopt the open-world assumption and assume that the views are sound. We do not go into details on the theoretical implications of these assumptions but refer to [1, 26, 84, 119] for more details.

Under these assumptions it is hard or even impossible to find an equivalent rewriting of a query in a PDMS. The best we can do is try to retrieve the maximal set of answers to a query, i.e., find a maximally-contained rewriting according to Definition 2.6.2, which will often involve a union of several queries over the sources.

Definition 2.6.2 (Maximally-Contained Rewriting)**.** *Let Q be a query, $\mathcal{LA}$ a query language, and $\mathcal{V} = \{V_1, \ldots, V_m\}$ a set of view definitions. The query Q' is a maximally-contained rewriting of Q using $\mathcal{V}$ w.r.t. $\mathcal{LA}$ if:*

- *Q' is a query in $\mathcal{LA}$ that refers only to the views in $\mathcal{V}$,*
- *Q' is contained in Q, and*
- *there is no rewriting $Q_1 \in \mathcal{LA}$, such that $Q' \sqsubseteq Q_1 \sqsubseteq Q$ and Q_1 is not equivalent to Q'.*

Note that a rewriting is maximally-contained only with respect to a specific query language. Thus, it might be possible that a rewriting providing more answers exists in a more expressive language. In the following we will for convenience often refer to maximally-contained rewritings simply as rewritings.

An essential part of Definition 2.6.2 is *query containment*, i.e., to decide whether a query Q' is contained in another query Q. To decide if Q' is contained in Q finding a containment mapping is a sufficient condition [33, 122]:

$$Q' \sqsubseteq Q \Longleftrightarrow \exists \text{ containment mapping from } Q \text{ to } Q' \tag{2.4}$$

Definition 2.6.3 (Containment Mapping). *A containment mapping [33, 120, 122] from Q to Q' is a symbol mapping ($h : \mathcal{S}(Q) \mapsto \mathcal{S}(Q')$) such that:*

- $\forall e \in \mathcal{E}(Q) : h(e) \in \mathcal{E}(Q')$
  there is a mapping for each exported variable e in query Q to an exported variable in Q',
- $\forall c \in \mathcal{C}(Q) : h(c) = c$
  each constant c in Q is mapped to the same constant in Q',
- $\forall l \in \mathcal{L}(Q) \exists l' \in \mathcal{L}(Q') : h(l) = l'$
  each literal l in Q is mapped to at least one literal l' in Q', and
- $\mathcal{P}(Q') \Longrightarrow \mathcal{P}(Q)$
  the conditions in Q' imply the conditions in Q.

Consequently, in order to obtain the maximal result set, the rewriting must contain mappings for all relations and attributes referred to by the query. The information we need for finding these mappings is provided by the view definitions. Thus, the main problems that remain to be solved by a rewriting algorithm are:

- to decide if a rewriting actually provides a mapping for all symbols used in the query and
- to ensure that the conditions in the query and in the views of the rewriting are not contradictory.

2.6.3 Query Rewriting Algorithms

On the previous pages we have limited our considerations to theoretical aspects of query rewriting. With this theoretical background, let us now review some state-of-the-art query rewriting algorithms. For this purpose, consider again a data integration system in which mappings are represented in LAV-style and queries are formulated as conjunctive queries.

There are three main rule-based algorithms for LAV-style mappings proposed in the literature dealing with conjunctive queries: the Bucket Algorithm [74,123,124], MiniCon [162,163], and Inverse-Rules [54,55]. Inverse-Rules basically inverts the information given by the view definitions to build rules describing how to extract records from the data sources and insert them into global relations to answer a query. Unfortunately, data joined locally is separated in the global relations. If there is the same join condition in both query and view, the records have to be recomputed and joined again. To avoid this and to obtain a more efficient rewriting, the inversed rules have to be unfolded.

The other two algorithms are based on another principle. They consider the subgoals contained in queries and views and use them to determine which combinations of views would not yield empty result sets – a subgoal is contained in the body of a query or view definition and consists of the reference to a relation with its attributes and possibly existing predicates. The first step in the Bucket Algorithm is to create one bucket for each subgoal contained in the query. As views are defined as queries, they contain subgoals as well. Thus, these subgoals can be used to decide whether a view (and consequently a source peer) provides relevant data or not. For this purpose, the subgoals of the views are compared to the subgoals of the query. Whenever a containment mapping from a view

subgoal to a query subgoal exists, the view is inserted into the bucket corresponding to the matching query subgoal. To reformulate the whole query, the Cartesian product of the views in the buckets is built. This Cartesian product may be rather large and may still contain a lot of combinations that would lead to empty answers, e.g., because of missing attributes for joins or conflicting predicates. In order to find rewritings, the algorithm performs a query containment test for each candidate combination. All candidates that do not pass the containment test are pruned. This is the main disadvantage of the Bucket Algorithm with regard to performance.

MiniCon is an advanced version of the Bucket Algorithm. It starts alike by creating buckets for query subgoals and sorting views into them. For each view definition the algorithm determines which query subgoals can be answered by the view. After having found a partial mapping from the query to the view, the join predicates of the query are considered in order to find out which additional set of view subgoals is needed to rewrite the whole query. This set and the mapping information is called a MiniCon Description (MCD). In the second phase, the MCDs are combined and query rewritings are created. Compared to the Cartesian product of the buckets in the Bucket Algorithm, fewer combinations of MCDs have to be considered. In MiniCon a portion of the work done in the second phase of the Bucket Algorithm is shifted into the first phase when building the MCDs. Thus, views that cannot be combined because of missing attributes for join conditions are detected at an early stage.

Example

To illustrate the process of query rewriting with an example, let us consider how the Bucket Algorithm rewrites a query. Assume we have the global schema:

```
writer(fullname, nation)
books(title, author, publicationDate)
```

and two sources:

```
S1:  item(title, writer, nationality, year), year > 1900
S2:  authors(name, nation)
```

The mappings `V1` and `V2` in LAV-style for the two sources `S1` and `S2` are defined as:

```
V1(t, w, n, y) :- writer(w, n),
                  books(t, w, y),
                  y > 1900
V2(n, nat) :- writer(n, nat)
```

Further assume we are given the following query in the global schema:

```
Q(t, a, n) :- writer(a, n), books(t, a, 1954)
```

Q selects *author*, *title*, and *nation* entries of all books published in 1954. In order to answer Q, we need to rewrite it so that the data of the sources can be queried. The Bucket Algorithm identifies two subgoals in the query and therefore creates two buckets:

```
writer(a, n)           books(t, a, 1954)
```

Afterwards, the view subgoals are compared to the query subgoals. `V1` is inserted into both buckets whereas `V2` is only inserted into one bucket. We obtain:

writer(a, n)	books(t, a, 1954)
V1(t, a, n, y)	V1(t, a, n, 1954)
V2(a, n)	

The bucket corresponding to `writer(a, n)` contains views `V1` and `V2` because the bodies of both views contain subgoals for whom variable mappings have been found:

```
V1:  (a -> w, n -> n)
V2:  (a -> n, n -> nat)
```

Note that `S1`'s predicate `year > 1900` does not contradict Q's predicate `year = 1954`. That is why `V1` was inserted into the bucket corresponding to `books(t, a, 1954)`. After having built the Cartesian product, we obtain:

```
q1(t, a, n) :-     V1(t, a, n, y),
                   V1(t, a, n, 1954)
```

and

```
q2(t, a, n) :-     V2(a, n),
                   V1(t, a, n, 1954)
```

`q1` can still be optimized by unifying the constant `1954` and the variable `y` so that `V1` occurs only once in the body of `q1`. In the last step, we need to check whether these combinations yield any contradictions. If, for example, `V1` contained a predicate `w` = "Charles Dickens" and `V2` a predicate `n` = "J.R.R. Tolkien", then `q2` would have an empty result set and would therefore be pruned. However, in our example there are no such contradictions so that the rewriting result is the union of two conjunctive queries:

```
q1(t, a, n) :-     V1(t, a, n, 1954),
UNION
q2(t, a, n) :-     V2(a, n),
                   V1(t, a, n, 1954)
```

2.6.4 Applicability to PDMSs

In this section we have reviewed state-of-the-art algorithms for query rewriting in the presence of LAV-style mappings. Although we can use these algorithms to rewrite queries in PDMSs, they lack the support of query operators such as top-N and skyline. As in particular the introduction of these operators into PDMSs is the focus of this dissertation, we need to develop an appropriate extension to these algorithms that considers these operators.

2.7 PDMS Implementations

In the previous sections we have highlighted the components that are relevant to the objective of this dissertation, i.e., to efficiently process rank-aware queries in PDMSs. Some of the approaches we apply, especially rank-aware query operators and routing indexes, have not been developed for PDMSs or data integration systems and are therefore not considered by any other state-of-the-art PDMS implementation. In this section, we discuss some of the most important PDMS implementations proposed in the literature. The main aspects that distinguish these implementations are sketched in Table 2.1.

Data Model	RDF – XML – Relational
Query Language	Conjunctive Queries – XQuery – XPath – SQL
Mappings	LAV – GAV – GLAV – Data Schema Interplay – Mapping Tables
Query Processing	Flooding – Index – Statistics

Table 2.1: Characteristics of PDMSs and their Spectrums of Values

The first aspect that distinguishes PDMS implementations is the data model a system works with, e.g., RDF, XML, or relations. In dependence on the data model, systems vary in the query languages they use to formulate queries. Some systems are based on datalog, others use XQuery-based languages, XPath, or SQL. Another distinction is made by the mappings that are supported. As mentioned above, mappings can be defined for example in LAV, GAV, or GLAV style. Some systems use special kinds of mappings that do not fit into these basic categories – we sketch some of them below. A final aspect we want to emphasize is that systems also vary in their query processing strategies. Many systems follow a simple flooding approach, i.e., forwarding the query to all the peers in the network – only pruning peers on schema-level when their schemas indicate their irrelevance. Some systems additionally apply indexes or statistics to prune peers on data-level as well. However, to the best of our knowledge none of them yet considers the application of routing indexes in the sense that we propose in this dissertation. To illustrate the differences between PDMS variants and our approach, the following pages discuss the most important PDMS implementations with respect to the characteristics of Table 2.1.

For the sake of completeness, it should be mentioned that there are also some systems dealing with the problem of schema heterogeneity in P2P systems, e.g., Edutella [50,145] relying on a HyperCuP [174] overlay topology. Therefore, the literature often refers to them as schema-based P2P systems. As PDMSs are also referred to as schema-based P2P systems, both kinds of systems are often classified the same [21] although they work on different levels. Whereas PDMSs provide algorithms for query rewriting, Edutella only indexes different schemas and allows queries to refer to elements that are part of different schemas. The query is not rewritten but simply routed to peers providing the referenced RDF schema elements that the original query refers to. Thus, because of the lack of query rewriting, we do not discuss these systems in more detail in this dissertation.

2.7.1 Piazza

Piazza [85, 188] is the most prominent peer data management system. The underlying network of peers is assumed to be a standard P2P system. However, the authors emphasize that other architectures such as a super peer network can be supported as long as there is a network of peers wanting to share semantically similar data.

Schema Heterogeneity Piazza aims to combine the two data integration formalisms LAV and GAV into one system. Data can be provided either in XML or RDF format. Queries are formulated in XQuery. Piazza knows two types of schema mappings: peer descriptions and storage descriptions. The former relate two or more peer schemas (schemas that peers publish to make their local data accessible to other peers) whereas the latter relate peer schemas and stored schemas (schemas of the actual data provided by the peers). There are two kinds of peer descriptions: equality descriptions ($Q_1(P_1) = Q_2(P_2)$ with Q_1 and Q_2 being conjunctive queries with the same arity and P_1 and P_2 being sets of peers) and inclusion descriptions ($Q_1(P_1) \subseteq Q_2(P_2)$) – equality can always be regarded as two inclusions. Thus, a mapping statement specifies a semantic mapping by stating that evaluating Q_1 over the peers P_1 will always produce the same answer (or a subset in the case of inclusions) as evaluating Q_2 over P_2.

Query Rewriting Queries are formulated as conjunctive queries and are rewritten in a centralized fashion. For this purpose, two techniques with respect to GAV and LAV mappings are used: *query unfolding* and *query answering using views* (MiniCon [84]). Both techniques are combined and applied recursively on the result using peer descriptions until no more rewriting is possible – applying the rule-goal-tree approach of [86]. Then, the storage descriptions are used for the final step of reformulation. The result of this rewriting process is a query on stored relations only (i.e., relations included in the storage descriptions).

Query Processing and Indexing In Piazza basically all available peers are contacted and thus participate in processing a query. The only possibility for a peer to get pruned (i.e., excluded from query processing) is when the mapping to the peer indicates that it cannot contribute relevant results. However, facing queries restricted to only a portion of the whole data space, it is worthwhile to identify peers that do not provide data in the queried range and to prune them as well. This problem can be solved by building an index. Therefore, Piazza offers a centralized index representing a summary of the data stored at the peers. Each peer participating in the system uploads a summary of its local data to a central instance (index engine) and refreshes it periodically. Users perform searches by submitting queries to the index engine (that also knows all peer mappings). The index consists of objects d that contain sets of attribute-value pairs of the following form: $d ::= [A_1 = v_1, A_2 = v_2, \ldots, A_n = v_n]$ where $A_1, \ldots, A_n$ are attributes and $v_1, \ldots, v_n$ are atomic values or patterns (with wildcards). The number of attributes may differ from one summary to the next so that the list of attributes does not have to be known in advance. Using the peer descriptions, relationships between attributes can be derived and queries of the form $q = [B_1 = w_1, B_2 = w_2, \ldots, B_p = w_p]$ with attributes $B_1, \ldots, B_n$ and constants $w_1, \ldots, w_n$ are supported.

2.7.2 Hyperion

Hyperion [169] names itself a Peer Database Management System (PDBMS) envisioned to be a conventional DBMS augmented with a P2P interoperability layer. This layer is built on top of a JXTA framework[1] [73, 149], which is used for communication between peers. Enabling communication and data exchange between peers results in a logical P2P network that might change over time. In this network peers providing similar data in a domain may form interest groups. Data is managed by conventional relational database management systems and queries are formulated in SQL. Each peer has its own DBMS and defines an independent access interface. The interoperability layer provides the peers with functionalities which enable data sharing.

Schema Heterogeneity Hyperion uses a special way to represent mappings - it applies mapping tables [109] listing pairs of corresponding values to search for data residing on different peers. Mapping tables represent expert knowledge and are typically created by domain specialists. More formally, a mapping table is a relation over the attributes $X \cup Y$, where X and Y are non-empty sets of attributes from two peers. Mapping tables explicitly express heterogeneity on data level – relating one set of attribute values to others and thus relating records. They can also be used to express constraints on information exchange between peers [109]. In general, a mapping table $m[X \cup Y]$ encodes not only data associations but also attribute correspondences between the set of attributes X and Y [135]. Furthermore, *translation rules* describe the relationships between corresponding attributes in terms of structure, format, and data values.

Query Processing and Rewriting Whenever there are corresponding entries in the mapping tables, a query is forwarded to the corresponding peer. Using this concept of representing mappings and constraints, specialized techniques of query rewriting are required [108]. We do not go into details as they are only useful in conjunction with mapping tables and are thus not relevant to this dissertation.

2.7.3 System P

System P [170, 171] is a PDMS that has been developed and published only recently. It uses, shares, and rewrites relational data and peers communicate via JXTA. For experimental purposes System P provides a PDMS generator: based on a reference schema this generator creates a given number of peers, local sources, heterogeneous schemas, and mappings. The data of the peers is either generated automatically or provided by the user in the predefined reference schema.

Schema Heterogeneity and Rewriting Just like Piazza, System P uses both local-as-view (LAV) and global-as-view (GAV) mappings to describe correspondences between schemas of neighboring peers. Mappings as well as queries are formulated as conjunctive queries using datalog rules. Though System P adopts the rule-goal-tree approach of [86] – also used by Piazza – it does not create a global plan that can be optimized. In contrast

[1] http://www.jxta.org

to Piazza, System P implements query rewriting in a completely decentralized way so that based on the given mappings a peer receiving a query rewrites it using only local information.

Query Processing, Indexing, and Routing For query processing System P provides a budget-driven approach that enables peers to exchange queries with associated budgets and resulting record sets. The main idea is to prefer peers and mappings that promise large partial result sets and mappings with low information loss. To prepare such decisions, each peer ranks all of its outgoing peer mappings (i.e., neighbors) according to the potential amount of data returned. This is achieved by estimating result cardinalities using multi-dimensional histograms [3] – applying query feedback to keep them up-to-date. The given budget is split up and assigned to the rewritten queries in accordance to the estimated information loss and result cardinality.

2.7.4 HePToX

HePToX [17,18] is a peer-to-peer database system that deals with XML data and heterogeneity. The outstanding feature of HePToX is that the user can specify correspondences by using a graphical interface to draw a set of visual annotations. Mapping rules are then derived automatically from these annotations so that queries can be rewritten.

Schema Heterogeneity Mappings are represented as datalog-like rules adapted to tree structured data. They are derived automatically from 1-1 correspondences defined by the user using the graphical interface. The basis for such annotations are DTDs representing schemas.

Query Processing and Rewriting Queries in HePToX are restricted to a subset of XQuery: queries that are expressible as joins of tree patterns [9]. A tree pattern is a rooted tree where nodes are labeled with variables. These variables may be constrained, e.g., on a value: $\$y/text() =$ "123". Queries are rewritten similar to Piazza applying a simplified version of an algorithm for answering queries using views [122].

2.7.5 coDB

Another approach worth mentioning in this context is coDB [63, 64]. It considers networks of peers with relational data. Such peers are interconnected by means of GLAV coordination rules (inclusions of conjunctive queries) and communicate using JXTA. The problem coDB aims to solve is to replicate data such that each peer may answer a query correctly with respect to initial data placement and mappings. For simulations a so-called super peer, which is connected to all participating peers, is introduced. During the simulation each peer collects statistics that are afterwards collected by the super peer and aggregated into the final statistics report. In contrast to many other systems, the authors considered the network's dynamic behavior, which means links between peers may be removed and added at runtime.

Query Processing and Rewriting Because of the GLAV definitions of mappings, rewriting in principle comes down to unfolding and does not pose any further problems. Queries in coDB are formulated by first order formulas in the language of the peer's local database. Query processing is implemented by first doing a global data update (issuing an update request to all neighboring peers) and then answering the query locally (using local replica). Each peer that receives an update request propagates it to all its neighbors (flooding). This process goes on until either a peer is found that has no more neighbors to propagate the query to or a peer receives the request more than once. The latter case is possible since cycles are not prohibited but identified by means of IDs assigned to the request messages.

2.7.6 Orchestra

Orchestra [78,79,189] is a *collaborative data sharing system* (CDSS) designed for the needs of bioinformatics and biomedical researchers. Building upon concepts developed in the Piazza project, Orchestra systems consist of a number of peers each of which having its own data schema. Each participant queries and manipulates the local database instance and may occasionally publish its database. Since in contrast to PDMSs researchers might have different viewpoints on what data is correct, modifications and updates are accepted according to some trust policy. This means that they might be accepted, rejected, or deferred to the future. This process of selectively importing updates (in particular those that do not conflict with existing data) is termed reconciliation. As data is modified at different sites, Orchestra publishes and propagates the updates to all sites that are willing to accept them.

Since in principle peers are allowed to have different schemas, updates need to be translated into other schemas such that peers receiving the updates are able to understand them. However, the preliminary results presented by Taylor et al. [189] assume a setup where all peers have the same data schema. Thus, the aspect of update translation remains future work. Recently, an implementation of Orchestra and a complete semantics for translating updates into a target schema, maintaining provenance, and filtering untrusted updates has been published [78].

A recently started project SHARQ (Sharing Heterogeneous and Autonomous Resources and Queries) [20] uses Orchestra as core engine. This project again focuses on biological data and aims at enabling biologists to find relevant information within a PDMS.

2.8 Conclusion

In this chapter we have reviewed related work relevant to the objective of this dissertation. We do not claim that the approaches and systems discussed so far represent a complete list of relevant related work. Rather, we meant to give the reader an overview of the state-of-the-art. The PDMS implementations we have discussed work well for the specific aspects they were built for. Unfortunately, they are very hard to extend to support efficient query processing for rank-aware queries. To extend PDMS functionality in this way, we identify the four main aspects/tasks discussed below.

Rank-Aware Query Operators Rank-aware query operators such as top-N and skyline represent important instruments for decision making. They are a very promising approach to reduce execution costs in distributed environments: because of their capability to reduce the size of the query result set, they also reduce the number of relevant peers (in conjunction with routing indexes) and consequently query execution costs. Moreover, applying the concept of approximation promises to reduce execution costs even further. To the best of our knowledge no existing PDMS implementation so far considers these aspects. Furthermore, the techniques for query processing and approximation we present for computing rank-aware operators are also novel for P2P systems and can be applied to those systems as well.

Query Processing The key to efficient query processing in distributed environments is to prune neighboring peers from consideration without unintentionally missing relevant data. For this purpose, a peer needs to know what data can be accessed via its neighbors. Routing indexes are appropriate structures that provide such information. The application of routing indexes, which has originally been proposed for unstructured P2P systems, enables a peer and its query processing strategy to consider result cardinalities for its routing decisions. Hence, peers can be ranked efficiently and pruned on data-level according to the expected number of records. However, most existing approaches for use in P2P systems do not capture attribute correlations. But without capturing them queries defined on multiple attributes (such as skylines) cannot be computed efficiently as the number of false routing decisions might be large. However, most implementations simply neglect the possibility of using additional information to prune peers on data-level. Although some of them use centralized indexes to overcome this problem, ignoring that in a PDMS there should be no instance with global knowledge, we argue that routing indexes should be used which adhere to the decentralized character of PDMSs. In this dissertation we propose not only efficient query processing strategies for rank-aware queries that exploit parallelism but also a novel variant of routing indexes enabling efficient query processing in PDMSs as well as in unstructured P2P systems.

Query Rewriting Because of schema heterogeneity, peers need to rewrite queries formulated on their local schema into the schemas of their neighbors. As most PDMS implementations apply the relational data model and conjunctive queries, the aspect of integrating rank-aware query operators goes beyond their capabilities. Thus, rewritings using existing approaches do not contain rank-aware operators. The portion of the data the rank-aware operator is defined on has to be transformed and sent to the query's initiator so that the rank-aware operators can be evaluated there. This causes not only high network load but also a high computational load at the peers. Hence, rank-aware query operators should be contained in rewritings as they are likely to prune most of the data already at the peer providing it. Although all PDMS implementations provide techniques for rewriting select-project-join queries, we cannot use them to rewrite queries containing rank-aware operators. Thus, this dissertation presents an appropriate extension.

Keeping Routing Indexes Up-To-Date As a query processing strategy that uses routing indexes relies on their correctness, we also need to find a solution to the problem

of keeping them up-to-date. There are multiple reasons that might require updating routing indexes: peers may join and leave the network or update their local data at any time. Most existing PDMS implementations do not consider this aspect. Since the query processing strategies that most systems propose come down to a simple flooding of the network, they actually do not have to pay attention to this problem: the only peers not queried are those whose schemas do not fit the query. But as soon as peers are pruned using information about attribute value ranges or cardinalities (routing indexes), some relevant data and thus answers might get lost if the indexes are not up-to-date. Thus, the system needs to detect such changes and apply strategies to keep the indexes up-to-date.

Even in P2P systems the problem of routing index maintenance has not yet been payed much attention to. The only strategies we are aware of are based either on propagating each single update through the whole network or on applying query feedback. As we will work out in this dissertation, both are no satisfactory solutions to the problem. Thus, this dissertation proposes and evaluates several strategies to efficiently keep routing indexes up-to-date.

Chapter 3

Model Definition

In the previous chapters we have already discussed the main characteristics of PDMSs. One of them is data heterogeneity. We stated that we limit our considerations to LAV-style mappings because in comparison to GAV-style mappings they represent the more complicated case. We argue that additionally integrating GAV-style mappings is possible but do not consider this option in this dissertation. In this chapter, we provide several basic definitions the following chapters build upon. This concerns the network model, mapping definition, and query formulation. Figure 3.1 highlights the steps of query processing in PDMSs that are affected.

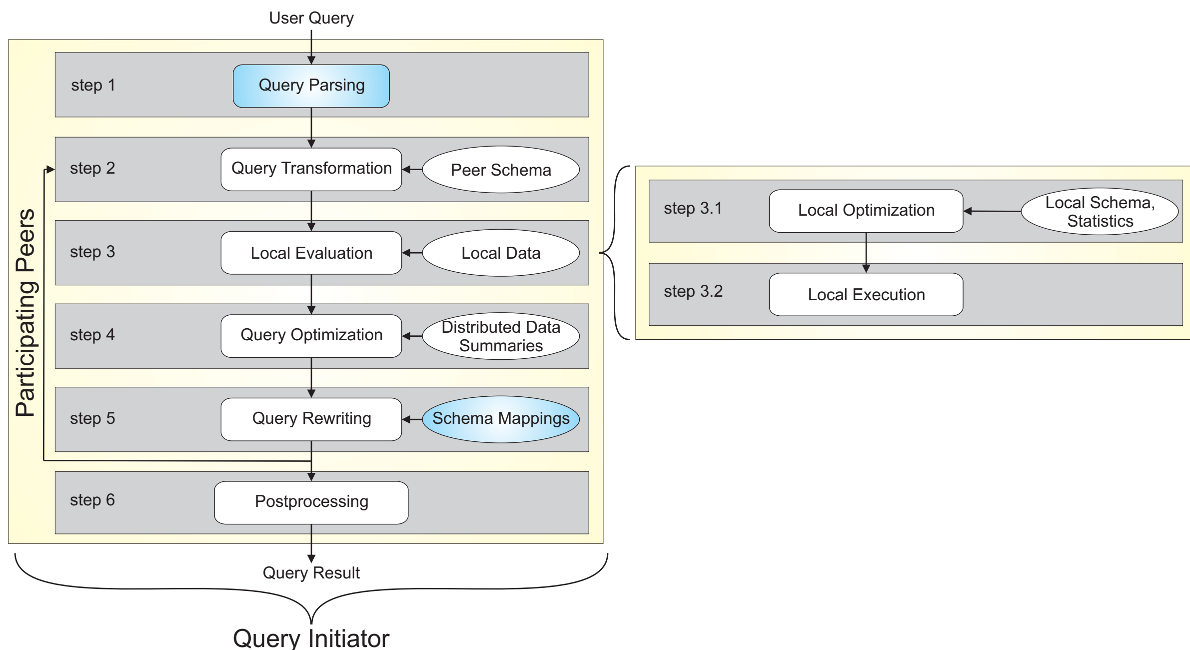


Figure 3.1: Query Processing in PDMSs - Query Formulation/Parsing and Mapping Definition

3.1 Network Model

As already mentioned in the introduction, we consider the network structure of a PDMS to be an unstructured P2P system (Definition 1.1.1). As native data format for our system we make use of XML because of its immense popularity for many applications that exchange data. Besides, most database systems offer the possibility to export data in XML format so that also non-XML data sources can participate and make their data accessible with low effort. Thus, peers with other native data formats are assumed to use wrappers and export their data in XML. There is no common agreement on connections (communication links) between peers and clustering. Although clustering peers according to their similarity in terms of schema and/or data would be beneficial, there is no directive to do so. Thus, we assume that the network structure is implicitly defined when peers join the network by establishing mappings and communication links to other peers in the system. We do not provide any means to automatically change the network structure and peers may have an arbitrary number of neighbors that they have established mappings to.

As we use LAV-style mappings, mappings are directed and expressed by views. Hence, in the following we often use the two terms mapping and view definition as synonyms. All mappings are stored locally at the peers so that a mapping is only known to the peer owning it. There is no global instance that might assist in rewriting the query. As a consequence, global reasoning and optimization is impossible. In contrast, at each peer the query is rewritten in a completely independent process. It is likely that when forwarding a query from peer A to peer D, all peers on the routing path between A and D (e.g., B and C) have to rewrite the query repeatedly into different schemas.

Figure 3.2 illustrates an example network consisting of six peers. The data the peers share in this example describes libraries, authors, and books. The schemas in the network differ in the naming of attributes, in the way the data is structured, and in the data the peers provide (e.g., some peers provide information about authors and their books whereas others focus on libraries only). Each data record (data object, data item) adheres to the schema of the peer providing it. Nesting of XML structures is expressed by indentation and substructures that might occur multiple times are marked with "*". Arrows between peers indicate the existence of a mapping between them. The direction of an arrow indicates the direction of the mapping. In our example there is an arrow and thus a mapping from P_0 to P_2 but not vice versa. This means P_0 knows a mapping to P_2 and thus can rewrite queries into the schema of P_2 but P_2 does not know a mapping to the schema of P_0 and thus cannot rewrite queries into the schema of P_0. In this example, P_0 holds mappings to almost all other peers in the system. Thus, this network comes relatively close to a standard data integration scenario with P_0 as mediator. However, as it is a PDMS, queries can be issued at any peer in the system. Furthermore, peers might leave the network at any time and peers joining the network do not have to establish mappings to P_0.

3.2 Mapping Definition Language

Let us consider an example LAV mapping that P_0 might use to rewrite queries from its local schema into the schema of P_1 (Figure 3.4). Similar to Piazza [85] our mapping

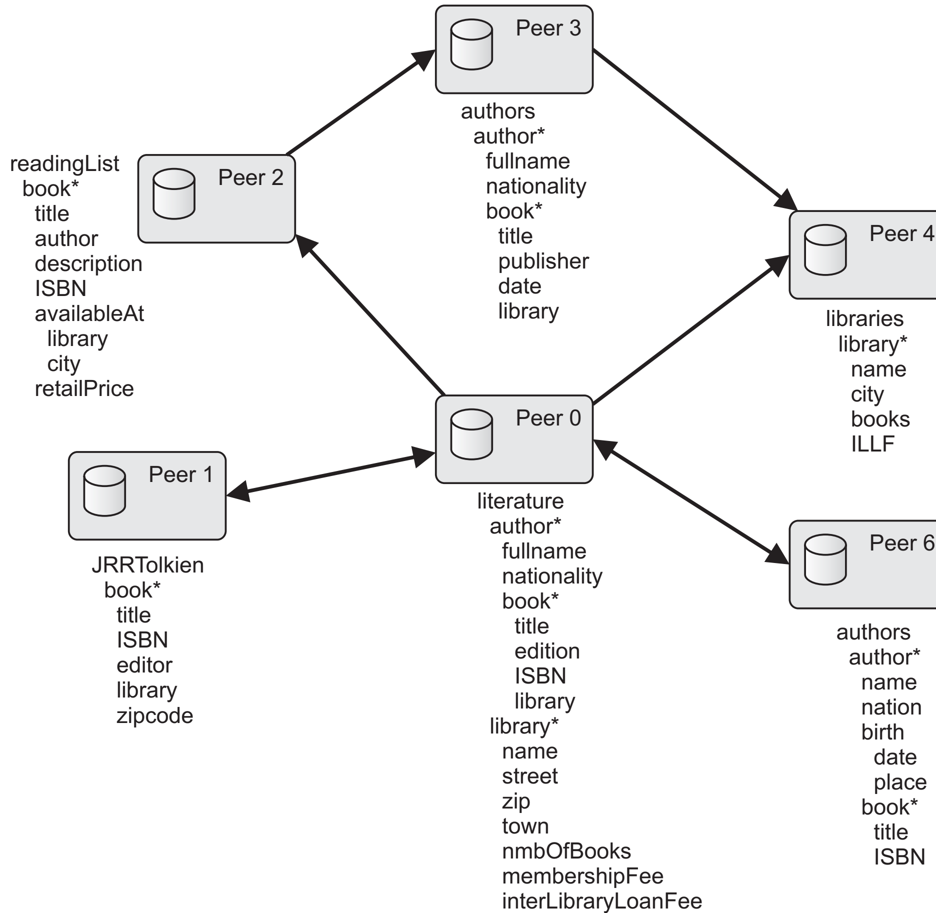


Figure 3.2: Example Network

language [94, 165] makes use of XQuery-like blocks. A view definition is well-formed and uses the XML structure corresponding to the schema of the neighboring peer (receiver schema) as basis (P_1's schema in our example).

Figure 3.3 illustrates the schema correspondences between the two schemas. The lower part of Figure 3.4 shows the complete mapping definition that P_0 (rewriter) uses to rewrite queries into the schema of P_1 (receiver) – Table 3.1 summarizes the terminology we will use in the following.

First, we need to define a set of context nodes. These are nodes contained in the schema of the rewriter and in the schema of the receiver corresponding to relations in the relational data model. A receiver context node is defined in the mapping definition by an XML attribute named *context*, which is assigned to an element in the receiver schema and therefore defines this element as the receiver context node. In our example the context attribute is assigned to an XML node `book`, which is part of P_1's local schema. This declares `book` as the receiver context node. The context attribute also defines the rewriter context nodes as well as variables. In our example the following variables are defined: `$a=literature/author`, `$b=$a/book`, and `$l=literature/library`. These variable declarations define the following rewriter context nodes: `literature/author` and `literature/library`.

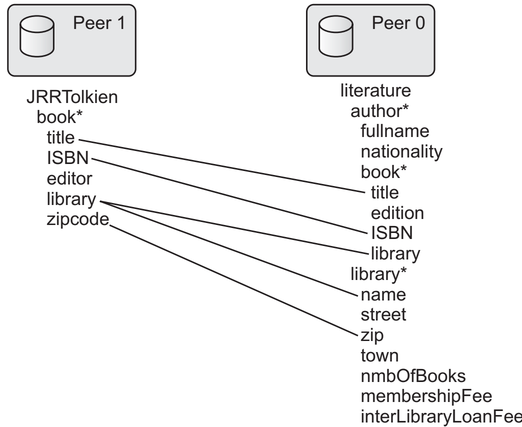


Figure 3.3: Correspondences between the Schemas of Two Peers (P_0 and P_1)

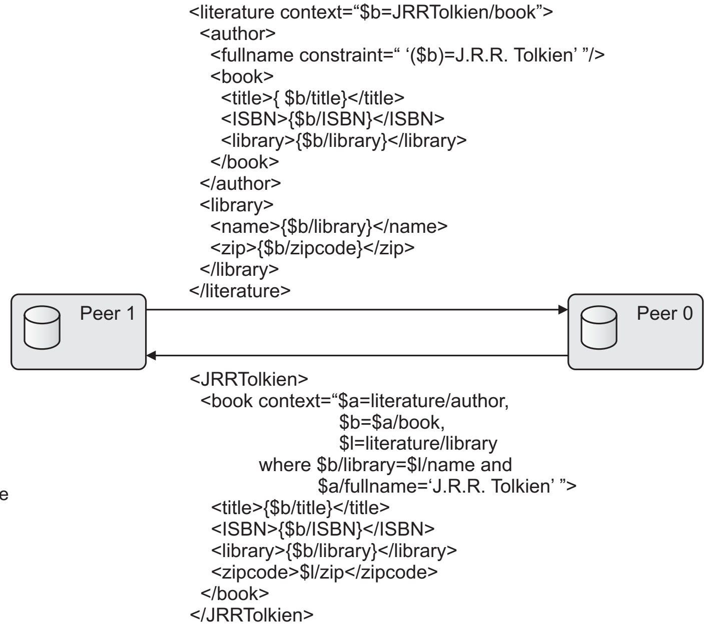


Figure 3.4: Mappings between Two Peers (P_0 and P_1)

Second, we define predicates to describe how the data is stored at the receiver. They are also encoded within the context attribute. Our example mapping defines two predicates `$a/fullname='J.R.R. Tolkien'` and `$b/library = $l/name`. The latter expresses a join and indicates that local structures of the rewriter are stored in a joined format at the receiver. The first predicate in our example involves a constant and indicates that the receiver's local data is restricted to information about J.R.R. Tolkien. In general, predicates have a restrictive effect and encode special conditions the rewriting process has to take care of, i.e., when rewriting a query these conditions have to be compared to predicates of the query and other views. Predicates can also be used to encode additional information. In our example, P_1 only stores data about books of J.R.R. Tolkien. However, this information is not encoded anywhere in its schema, rather it represents additional information that P_0 might use in a view definition. Thus, the view definition contains the predicate: `$a/fullname='J.R.R. Tolkien'`.

Finally, a mapping definition also needs to encode the correspondences themselves. This is realized by relating symbols (referring to a schema element of the rewriter) with schema elements of the receiver. In our example `<zipcode>{ $l/zip }</zipcode>` expresses the correspondence between P_1's element `JRRTolkien/book/zipcode` and P_0's element `literature/library/zip`.

Finally, let us make some final remarks on view definitions. Nesting of views is not allowed, i.e., in the descendent axis of an element with a context attribute there must not exist any other node with a context attribute. However, the view definition might still contain multiple context attributes if they are on the same hierarchical level. For example, in the mapping that P_0 holds for P_1 (lower part of Figure 3.4) it would be possible that the view definition contained a second context attribute assigned to a sibling node of `JRRTolkien/book`. In this case, each context attribute would declare a view on its own,

Term	Example	Definition
rewriter	P_0	*peer that rewrites the query – query is formulated in the schema of the rewriter*
receiver	P_1	*peer that receives the query – query is rewritten into the schema of the receiver*
rewriter schema	P_0's schema	*schema of the rewriter*
receiver schema	P_1's schema	*schema of the receiver*
rewriter context node	`literature/author`	*XML node in the rewriter schema that has been assigned to a variable*
receiver context node	`JRRTolkien/book`	*XML node in the schema of the receiver that defines the root node of all insertions*
variable/symbol	`$a`	*variable representing a rewriter context node*
symbol	`$a/book`	*variable in conjunction with a path expression*
exported symbol	`$b/title`	*symbol that is exported into the receiver schema*
constant	`J.R.R. Tolkien`	*invariable value*
condition/predicate	`$b/library = $l/name`	*predicate using symbols, constants, and arithmetic comparison operators*
constraint	(\$b)='J.R.R. Tolkien'	*predicate encoding additional non-structural information in a view definition*

Table 3.1: Terminology

i.e., P_0 would know two view definitions it could use to rewrite a local query into the schema of P_1.

In general, all predicates can make use of the following comparison operators: $<$, $\leq$, $>, \geq, =$. We explicitly exclude negation because of the problems this entails for rewriting [1]. If P_0 sends a query to P_1 it assumes that it only receives data corresponding to J.R.R. Tolkien. On the other way round, if a user formulates a query on the schema of P_1, he/she also assumes to retrieve only data on books of J.R.R. Tolkien. Thus, the queries that P_1 forwards to neighboring peers (e.g., to P_0) have to query data on books written by J.R.R. Tolkien. However, as there is no corresponding element in the schema of P_1, it cannot encode this information in its local schema. Consequently, a mapping from P_1 to P_0 (defined using the local schema of P_0 as basis) must contain additional information about the restriction on books written by J.R.R. Tolkien. For this purpose, a *constraint* may be integrated into the mapping. The upper part of Figure 3.4 shows the corresponding mapping. Remember that as a counterpart to this constraint P_0 has the following predicate in its view definition to P_1: `$a/fullname='J.R.R. Tolkien'`. In summary, view definitions consist of:

- variables defining context nodes (`$a`),
- symbols (`$a/book`),
- exported symbols ({`$b/title`}),
- constants (`'J.R.R. Tolkien'`),
- conditions corresponding to predicates (`$b/library = $l/name`), and
- constraints (`($b)='J.R.R. Tolkien'`).

Thus, for each view V we distinguish between:

- the set of variables $\mathcal{L}(V)$,
- the set of symbols $\mathcal{S}(V)$,
- the set of exported symbols $\mathcal{E}(V)$,

- the set of constants $\mathcal{C}(V)$,
- the set of conditions/predicates $\mathcal{P}(V)$, and
- the set of constraints $\mathcal{T}(V)$.

Although we are aware that mappings might change (replaced or improved) over time, we do not yet consider this aspect in our system. Future work might consider this aspect and provide techniques for automatic schema matching [167] in conjunction with schema composition and adaptation. In this context future work might also address the consequences of information loss originating from incorrect or incomplete mappings. However, for the time being we assume that mappings are correct and complete.

3.3 Query Formulation

In the introduction we have already formally defined the structure of a PDMS (Definition 1.1.1) and discussed the main steps of query processing in these environments (Section 1.2). The first step needs to be query formulation and parsing as highlighted in Figure 3.1. As we assume peers to provide data in XML format, XQuery is a good option to formulate queries. However, it lacks the support of rank-aware query operators. Basically, we could define an extension so that skyline queries might, for instance, be formulated as follows:

```
for $l in fn:doc("libraries.xml")//library
skyline of MIN $l/nmbOfBooks
           MAX $l/interLibraryLoanFee
return ...
```

In order to provide a high degree of flexibility and because it is not the contribution of this dissertation to define formal extensions to existing query languages, we do not introduce such an extension but regard queries on an algebraic level. Thus, queries are formulated using a set of algebraic operators as defined below. In future work our system might still be extended to support such extended query languages.

As building blocks of query formulation we make use of algebraic plan operators (POPs). POPs can be combined into POP trees. Each such tree represents one query. The result of a query is processed in a bottom-up fashion such that leaf nodes are computed first. A parent POP uses the output of its children as input. Both input and output are sequences of XML structures. The output of the tree's root node represents the answer to the query.

There are only a few basic rules that a query must adhere to. As mentioned above, the query representation corresponds to a tree structure, i.e., each POP has exactly one parent POP except the root node. The number of children depends on the POP's type. Leaf nodes of the tree are always *select* POPs because only these POPs can directly refer to the local data of a peer. Thus, they select the data that all POPs on higher levels operate on. They represent a starting point and define the selection expressions (XPath) that are evaluated on the peers' local data. The result of evaluating a select POP is a sequence of XML structures whose root nodes are obtained by evaluating the XPath expression on a peer's local data (select POP on leaf level) or on the output of its child POP. Table 3.2 summarizes the POPs we consider in this dissertation.

	POP	symbol	alternative symbol	#children	parameters
user-level	select/project	σ	`select`	0/1	1 XPath expression
	union	$\cup$	`union`	2	—
	join	$\bowtie$	`join`	2	1 condition
	skyline	Φ	`skyline`	1	$\geq$ 2 ranking functions
	topn	Φ	`topn`	1	integer n, 1 ranking function
	construct	κ	`construct`	1	1 construct expression
system-level	remote query	r	`remote`	1	neighborID

Table 3.2: List of Algebra Operators

There are only two POPs that require two child POPs: the *union* POP, which merges the two result sets of its child POPs, and the *join* POP. The *join* POP receives two sequences of XML structures as input and joins any two XML structures originating from different sets that fulfill the join condition. All the other POPs may only have one child POP, except a select POP on leaf level, which has none. The two matching XML structures are combined by the join POP with an enclosing XML element named `<pair>`. Let us assume a join has the following two input sets and the join predicate `literature/library/zip = book/zipcode`.
First input set:

```
<literature>
  <library>
    <zip>D-98693</zip>
    <town>Ilmenau</town>
  </library>
</literature>
```

Second input set:

```
<book>
  <title>The Lord of the Rings</title>
  <zipcode>D-98693</zipcode>
  <ISBN>0261103253</ISBN>
</book>
```

Based on this input the join POP produces the following output:

```
<pair>
  <literature>
    <library>
      <zip>D-98693</zip>
      <town>Ilmenau</town>
    </library>
  </literature>
  <book>
    <title>The Lord of the Rings</title>
```

```
      <zipcode>D-98693</zipcode>
      <ISBN>0261103253</ISBN>
   </book>
</pair>
```

The join POP is the only operator that creates new elements and applies nesting automatically, i.e., for each pair of matching XML structures, a new structure with `pair` as root element and the two matching structures as children is created. We decided not to merge the matching structures because by keeping both of them, XPath expressions that are part of the definitions of operators on higher levels do not have to be adapted much – only the `pair` element needs to be considered. Furthermore, from an algorithmic point of view it is unclear how to correctly merge the two structures automatically. By using a `construct` POP (see below) the query, or the user respectively, can still define an operator with a description how to merge the structures correctly.

As the *select* POP is given an XPath expression as input, it performs not only selection but also projection and may also occur as an inner node of a query POP tree. The *construct* POP is used to restructure XML data according to the associated expression (renaming, nesting, and unnesting). This expression encodes the new structure that the input XML structures are to be transformed into. It defines the output structure with XML tags and the data that is to be inserted into those tags by path expressions referring to the input schema. Let us assume, for example, the following *construct expression* is assigned to a *construct* POP – nesting (in this example we use an element named `res`), unnesting, and labeling nodes are choices made by the user when formulating the query:

```
<res>
   <writer>{ author/name }</writer>
   <booktitle>{ author/book/title }</booktitle>
</res>
```

The result of an evaluation on the following document

```
<author>
   <name>J.R.R. Tolkien</name>
   <book>
      <title>The Lord of the Rings</title>
      <ISBN>0261103253</ISBN>
   </book>
</author>
```

would be:

```
<res>
   <writer>J.R.R. Tolkien</writer>
   <booktitle>The Lord of the Rings</booktitle>
</res>
```

For each input XML structure the writer's name and the title of his/her book is selected and transformed into another structure with `res` as outer element, which has two child elements `writer` (containing the author's name) and `booktitle` containing the title of the author's book. The result set of this *construct* POP contains one such XML structure for each input structure.

Top-N and skyline operators are represented by *topn* and *skyline* POPs – we use the same symbol for both operators as they both represent query operators of the same class (rank-aware operators). The *topn* POP expects an integer number n ($n \geq 1$) as input as well as a ranking function using standard mathematical operators as well as aggregate functions such as SUM, MIN, and MAX. The *skyline* POP needs at least two such functions as input because a skyline is always defined on multiple ranking functions.

In contrast to all other POPs the *remote query* POP cannot be used to formulate queries by the user. Therefore, Table 3.2 distinguishes between POPs on user-level and system-level. The remote POP is generated by the rewriting algorithm (Chapter 4) to denote those query subtrees that are forwarded to neighboring peers.

So far we have only defined queries as tree structures consisting of POP trees whose leaf nodes must be *select* POPs. However, this is only a representation, the same queries can be issued in text format and formalized according to Definition 3.3.1.

Definition 3.3.1 (Query). *A query Q is contained in query language L_Q generated by grammar G_Q, i.e., $Q \in L_Q(G_Q)$ with $L_Q(G_Q) := \left\{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\right\}$. Grammar G_Q is a quad-tuple (V,Σ,S,P), with*

- *V being the finite set of non-terminal symbols: $V = \{A, B\}$,*
- *Σ being the finite set of terminal symbols: $\Sigma = \mathcal{S} \cup \mathcal{O} \cup \mathcal{A}$, which consists of symbols for structuring $\mathcal{S} = \{(,), `,'\}$, symbols for query operators $\mathcal{O} = \{\texttt{select}, \texttt{union}, \texttt{join}, \texttt{skyline}, \texttt{topn}, \texttt{construct}, \texttt{remote}\}$, and symbols representing arguments of the operators $\mathcal{A} = \{condition, ranking, n, rankingList, constExp, nID\}$ as listed in Table 3.2,*
- *$S \in V$ being the start symbol: $S = A$, and*
- *P being the set of production rules:*

$$
\begin{array}{lcl}
A & \mapsto & (B) \\
B & \mapsto & \texttt{select}\ `path' \\
B & \mapsto & \texttt{select}\ `path'\ (B) \\
B & \mapsto & \texttt{union}\ ((B)\ (B)) \\
B & \mapsto & \texttt{join}\ `condition'\ (B)\ (B) \\
B & \mapsto & \texttt{skyline}\ `rankingList'\ (B) \\
B & \mapsto & \texttt{topn}\ `n'\ `ranking'\ (B) \\
B & \mapsto & \texttt{construct}\ `constExp'\ (B) \\
B & \mapsto & \texttt{remote}\ `nID'\ (B)
\end{array}
$$

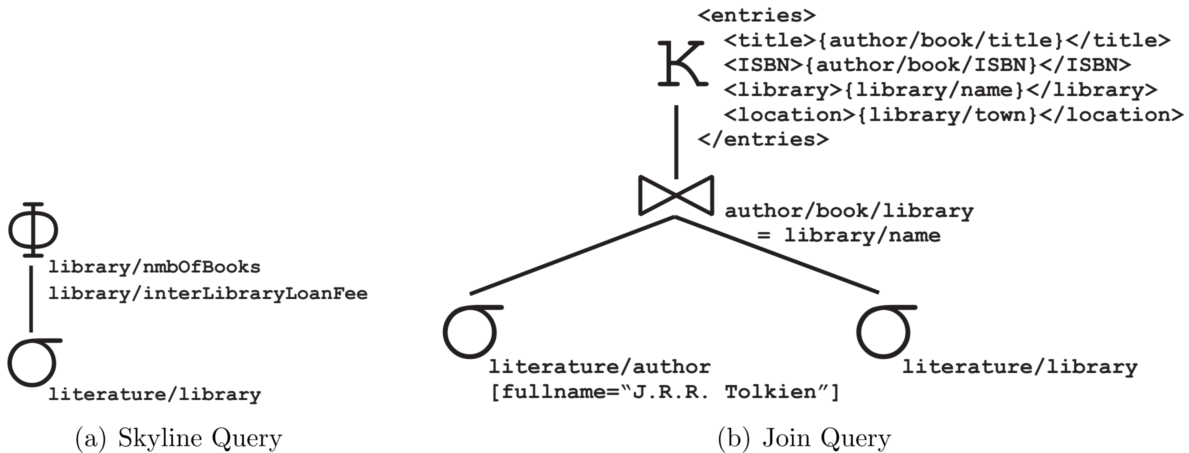
In this definition, *path* denotes an XPath expression, *condition* denotes an equality expression referring to schema elements of the operator's child nodes (denoted using path expressions), *ranking* denotes an annotated ranking function formulated on the child operator's schema elements, *rankingList* denotes a sequence of annotated ranking functions, *constExp* denotes an expression defining the schema that the input XML structures are to be transformed into, and *nID* denotes the ID of a neighboring peer.

Let us consider some example queries formulated in L_Q that might be issued at P_0 (Figure 3.2). First, let us formulate a query asking for the skyline on libraries that provide many books but charge a minimum inter library loan fee. The corresponding query is:

```
(skyline 'MAX("library/nmbOfBooks")'
         'MIN("library/interLibraryLoanFee")'
   (select 'literature/library')
)
```

In the following we will often depict the query in the more illustrative format of a POP tree. Figure 3.5(a) shows the POP tree corresponding to the skyline query defined above. For the sake of clarity, the figure only shows the schema elements the query is defined on. Another query issued at P_0 might ask for works of J.R.R. Tolkien and libraries that provide his books. Assuming the query also restructures the result records with a construct POP, the user might issue the following query (Figure 3.5(b) shows the corresponding POP tree):

```
(construct '<entries>
               <title>{author/book/title}</title>
               <ISBN>{author/book/ISBN}</ISBN>
               <library>{library/name}</library>
               <location>{library/town}</location>
            </entries>'
   (join 'author/book/library=library/name'
     (select 'literature/author[fullname="J.R.R. Tolkien"]')
     (select 'literature/library')
   )
)
```



(a) Skyline Query (b) Join Query

Figure 3.5: POP Tree Representation for Example Queries

3.4 Summary

In this section we have defined the environment that we are working with. We defined the network model as an unstructured P2P system with each peer providing data in XML format and maintaining LAV-style mappings to neighboring peers. Furthermore,

we have shown how to define mappings and introduced the basic terminology that we will use in the following. Another important issue we elaborated on in this chapter is how we assume queries to be formulated. All these aspects have a great influence on appropriate query rewriting algorithms. In particular the fact that we allow rank-aware operators to be contained in a query definition has a great influence on query rewriting and optimization as we will see in the following chapter.

Chapter 4

Rewriting Rank-Aware Queries for XML Data

After having completed the first step of query processing (Figure 4.1), the peer holds an appropriate POP tree representation of the query that refers to its local peer schema. In the second step, the query is transformed in consideration of the global schema (normalization, unnesting, simplification). In this dissertation, we do not go into details on this aspect but assume the POP tree given as input is optimized with respect to the techniques applicable in this step. Afterwards, in step three, the query is evaluated on the peer's local data. For simplicity, we assume the data is stored in the peer schema so that local query evaluation is straightforward. The peer can additionally apply local optimization techniques to minimize local execution costs. However, the local result can be used as additional information in the fourth step (query optimization), which applies a query processing strategy and utilizes distributed data summaries to identify neighbors providing relevant data. After these neighbors have been identified, the query is rewritten (step five) in consideration of the schemas of relevant neighbors only. As we will see later, using the local result as additional information in the fourth step is beneficial for evaluating top-N and skyline queries. For other query types, e.g., exact match queries or range queries, we can postpone the local execution step and execute it in parallel to query processing at neighboring peers.

Obviously, there are two possible orders of execution for steps four and five. If routing indexes are considered before rewriting, they have to be defined in the local schema of the peer. In doing so, the number of relevant neighbors is reduced so that less mappings have to be considered for rewriting. Alternatively, if rewriting is performed first, all neighbors and thus all mappings need to be considered by the rewriting process. Afterwards, some of the created subqueries are pruned if the routing indexes indicate their irrelevance.

In order to reduce rewriting costs, we decided to have peers consider routing indexes first and perform rewriting afterwards. However, we still begin our discussion with query rewriting because in the previous chapter we have already begun to discuss schema-related topics and rewriting is a prerequisite for the techniques we present for routing index construction and maintenance – just like queries, updates for routing indexes also need to be rewritten.

In this chapter we build upon the model defined in the previous chapter and present a distributed algorithm [94,165] for query rewriting (using LAV-style mappings) on XML

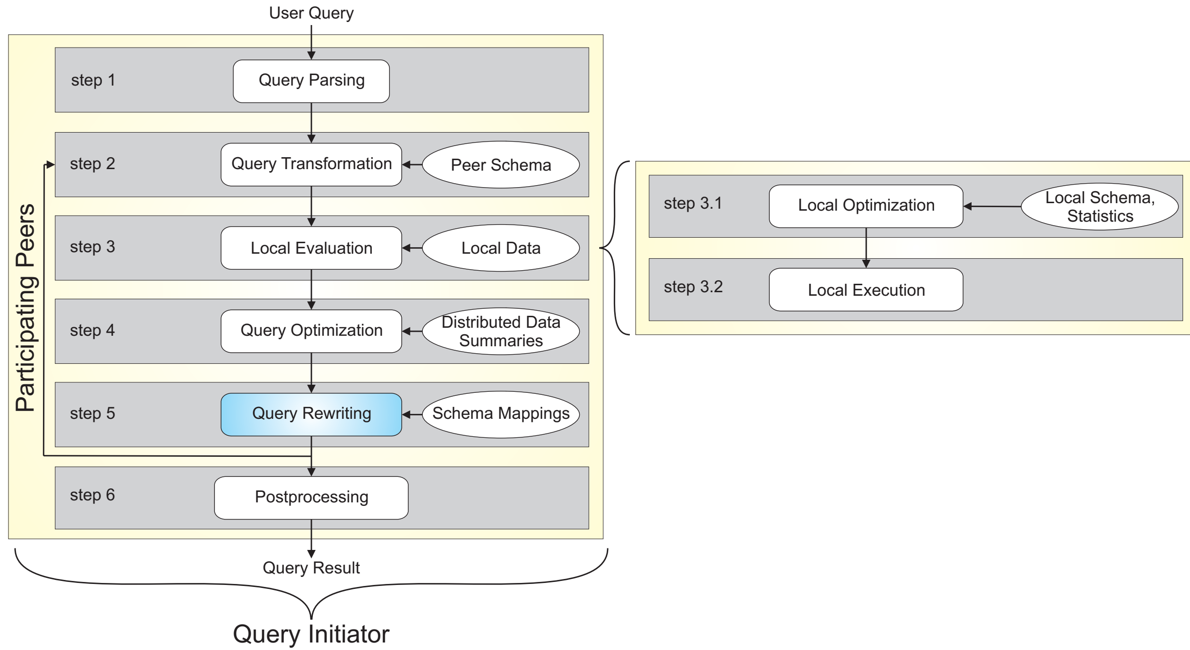


Figure 4.1: Query Processing in PDMSs – Query Rewriting

data that also considers rank-aware query operators instead of being restricted to select-project-join queries.

4.1 Subgoal Trees

In Section 2.6.3 we have studied the most important state-of-the-art query rewriting algorithms. The solution we propose with respect to the model and requirements defined in Chapter 3 is basically an extension and a combination of the Bucket Algorithm [74,123, 124] and MiniCon [162,163]. Our solution works on plan operator trees instead of datalog rules and pays attention to all query operators that have been introduced in Section 3.3 including skyline and top-N. Just like MiniCon but unlike the Bucket Algorithm, we pay attention to predicates at an early stage of the rewriting process. In contrast to the Bucket Algorithm, subgoals are sorted into the buckets, not views. As identifying and comparing subgoals is a crucial part of the rewriting algorithm, we need to discuss how to identify subgoals of queries and views as well as how to represent them before going into details on the rewriting algorithm itself.

4.1.1 Creating Subgoal Trees for Views

Working with XML data, rewriting has to deal with nesting and unnesting of elements or structures, respectively. Hence, it is beneficial to represent subgoals in a tree-like structure because nesting can be represented easily with trees. When a view contains multiple subgoals, each subgoal corresponds to a semantically independent structure such as a relation for relational data. Each subgoal contained in a view definition is represented by a subgoal tree as defined in Definition 4.1.1 – remember the terminology summarized in Table 3.1.

Definition 4.1.1 (View Subgoal Tree)**.** *A view subgoal tree T consists of a set of annotated nodes arranged in a tree structure according to the following rules:*

- *T has exactly one* root node *annotated with:*
  (i) the path to the rewriter context node and
  (ii) the path to the receiver context node
- *multiple* inner nodes *annotated with:*
  (i) the name of an element in the rewriter schema or
  (ii) the name of an element in the receiver schema
- leaf nodes *annotated with:*
  (i) the name of a text node in the rewriter schema,
  (ii) the name of a text node in the receiver schema or a constant corresponding to such a node, and
  (iii) optional predicates.

This structure will help us find a symbol mapping between rewriter and receiver schema elements. The root node of a subgoal tree contains the name or path of context nodes in the rewriter and receiver schemas – such a context element is comparable to a relation name in RDBMSs. Inner subgoal tree nodes represent the nesting of elements at the view's owner (rewriter) or at the receiver. Within a subgoal tree all symbols, or rather all schema elements (schema nodes) referred to by the symbols contained in the view definition, are represented.

In accordance with Section 3.2, we can determine for each subgoal s the set of corresponding symbols ($\mathcal{S}(s)$), the set of exported symbols ($\mathcal{E}(s)$), the set of constants ($\mathcal{C}(s)$), the set of conditions/predicates ($\mathcal{P}(s)$), and the set of constraints ($\mathcal{T}(s)$). Thus, all symbols in $\mathcal{S}(s)$, or rather the paths and schema elements referred to by them, are represented by nodes in the subgoal tree corresponding to subgoal s. All exported symbols in $\mathcal{E}(s)$ are represented by leaf nodes. Likewise, all constants $\mathcal{C}(s)$ and conditions $\mathcal{P}(s)$ are assigned to leaf nodes. Only constraints $\mathcal{T}(s)$ are not represented within the structure of the tree but assigned as additional characteristics to the subgoal tree.

As an example let us consider the view definition between P_0 and P_1 (Figure 3.4) that we have introduced in Chapter 3. There are two subgoals defined by the symbols `$a` (`literature/author`) and `$l` (`literature/library`). Each of these paths defines a rewriter context node and thus a subgoal tree. Both of them are depicted in Figure 4.2.

The algorithm that creates such subgoal trees with mapping definitions as input is sketched in Algorithm 1. At first, the algorithm finds all nodes in the view definition with context attributes (line 1) – *receiver context nodes*. For each context attribute a separate *view* is initialized (line 2). For all symbols defined in the context attribute we check (line 6) whether we need to create a new subgoal (lines 16–20) or add the information to an existing one (lines 7–14). We do not create a new subgoal if the symbol refers to another symbol or shares a common prefix path. In either case, nodes are created in the subgoal tree for all those elements occurring in the path of any symbol definition. Afterwards, we need to check and model the predicates defined in the context attribute. As a first step the *predicate factory* collects all predicates contained in the context attributes (lines 22–25), assigns them to the corresponding nodes in the subgoal tree if they have already been created. For each variable contained in a predicate the

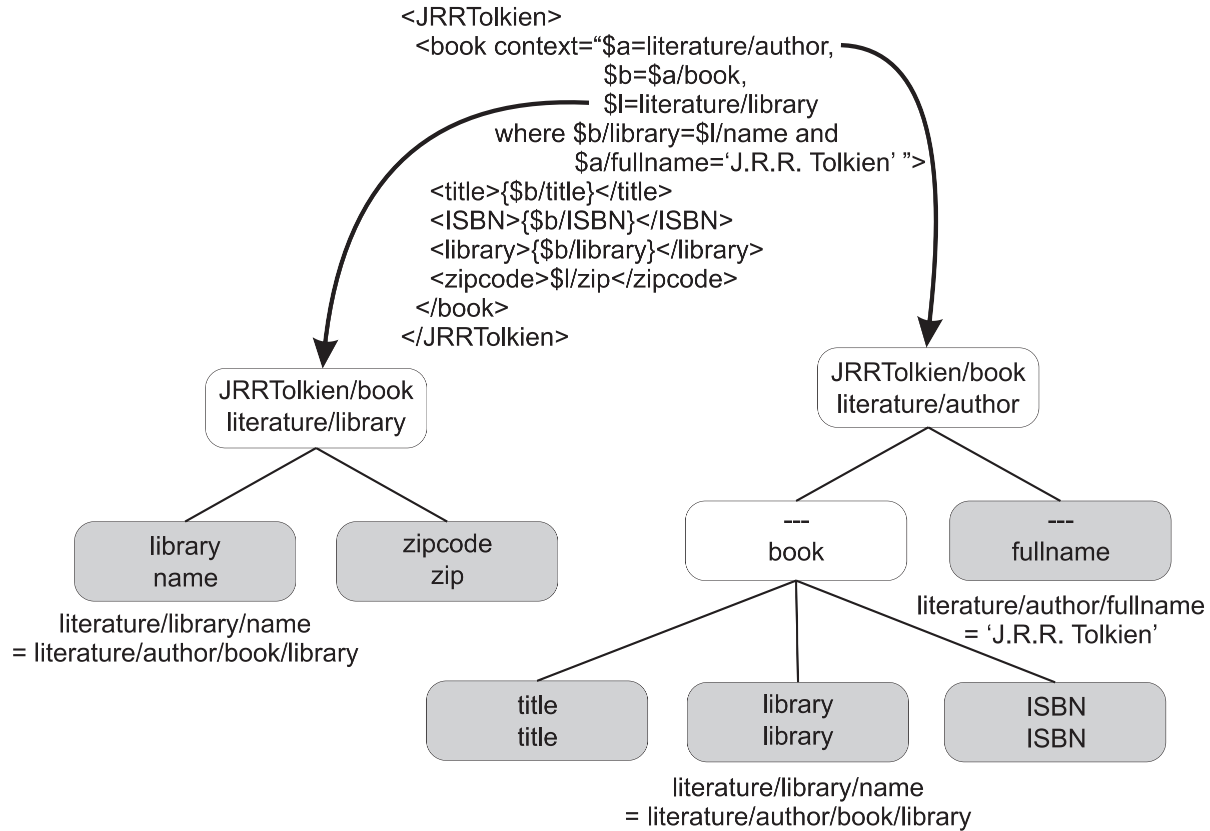


Figure 4.2: Creating Subgoal Trees for the View Definition from P_0 to P_1. *Each subgoal tree node has two annotations: a receiver schema annotation (first row) and a rewriter schema annotation (second row). Exported nodes are tagged (grey).*

factory resolves and stores the absolute paths, e.g., for `$b/library` the factory stores `literature/author/book/library`.

So far we have only created representative nodes for schema elements that have been mentioned in any symbol declaration in the context attribute. This means that all nodes of the subgoal trees so far only represent rewriter schema elements. The only exception is the root node that has been assigned both a rewriter and a receiver path (line 17). The receiver path of the root node is determined by the node in the view definition that contains the context attribute (e.g., `JRRTolkien/book`) and the rewriter path of the root node is determined by a symbol declaration (e.g., `literature/author`). In order to create representations for receiver schema nodes within the subgoal tree, we need to recursively traverse all child nodes of the view definition's receiver context node (lines 26–29). Method *createNodes* takes care of creating nodes for both receiver schema elements (descendants of the receiver context node in the view definition) and rewriter schema elements (referred to by exported symbols contained in receiver schema elements). When creating a leaf node in the subgoal tree (e.g., for `zipcode` (receiver) and `zip` (rewriter)), the node is marked as exported and the factory is asked if there exists any predicate that involves the currently considered rewriter schema element – in Figure 4.2 exported nodes are highlighted in grey. If so, the corresponding predicates are assigned to the leaf node of the subgoal tree. If the rewriter element is involved in a join, i.e., it is part of a

Algorithm 1 *createQuerySubgoalsView(viewDef)*

```
1: for all contextEl ∈ viewDef.getNodesWithContextAttribute() do
2:   view = new View();
3:   receiverPath = contextEl.getFullPath();
4:   contextClause = contextEl.getContextClause();
5:   for all variableBinding ∈ contextClause.getVariableBindings() do
6:     if variableBinding.getRewriterPath.startsWith('$') then
7:       /* if rewriterPath belongs to a subgoal that has already been created */
8:       node = findLastExistingNodeFor(rewriterPath);
9:       for all element in rewriterPath after node do
10:        /* create new nodes */;
11:        newNode = new Node(element.getName(). '');
12:        node.addNode(newNode);
13:        node = newNode;
14:      end for
15:    else
16:      /* create a new subgoal tree with a new root node */
17:      node = new Node(rewriterPath, recieverPath);
18:      subgoal = new SubGoal(node);
19:      view.add(subgoal);
20:    end if
21:  end for
22:  /* collect and assign predicates */
23:  for all predicate ∈ contextClause.getPredicates() do
24:    factory.createPredicatesInView(predicate);
25:  end for
26:  /* create nodes representing rewriter/receiver schema elements */
27:  for all child ∈ contextEl.getChildren() do
28:    createNodes(child, new Vector(), factory);
29:  end for
30:  /* treat the predicates related to not exported symbols at the end */
31:  treatLeftoverPredicates(factory);
32: end for
```

join predicate, it is necessary to clone all predicates that have been assigned to its join partner and assign them to the leaf node.

Now all predicates are assigned to exported symbols (i.e., leaf nodes in the subgoal tree). There is only one special case that we need to pay attention to: predicates not involving any exported symbols or predicates that refer to elements to which the receiver does not have a corresponding element, e.g., `$a/fullname = 'J.R.R. Tolkien'`. In this case, we need to create additional non-exported leaf nodes in the subgoal tree to represent such predicates (line 31).

Algorithm 2 provides more details on how method *createNodes* creates inner nodes in the subgoal tree. The initial input of the algorithm is a receiver node and the reference to the predicate factory. At first, the algorithm checks if a constraint is assigned to the input receiver node. If so (lines 1–4), this constraint is assigned to the whole subgoal tree rather than to the receiver node that is involved. If the current receiver node is an inner node, the node's name is stored into a list of node names (*nodeNames*) and *createNodes* is recursively invoked on its children (lines 5–10). This list of receiver node names contains all nodes that have been visited so far – representing nodes on the path between the receiver context node and the current node in the receiver schema.

If the currently considered node of the receiver schema is a leaf node in the view definition, it contains an exported symbol. The path of the exported symbol is extracted (line 13) and the predicate factory is asked, whether there are any join predicates involving this path (line 14). The reason is that both structures that are involved in a join have to be treated equally and predicates involving either one of them have to be assigned to

Algorithm 2 *createNodes(receiverNode,nodeNames,factory)*

```
1: constraint = receiverNode.getConstraint();
2: if constraint ≠ null then
3:    subgoal.setConstraint(constraint);
4: end if
5: if receiverNode.hasChildren() then
6:    /* if receiverNode is an inner node in the view definition */
7:    nodeNames.add(receiverNode.getName());
8:    for all child ∈ receiverNode.getChildren() do
9:       createNodes(child, nodeNames, factory);
10:   end for
11: else
12:   /* receiverNode is a leaf node in the view definition, i.e., exported symbol found */
13:   rewriterSymbol = receiverNode.getExportedSymbolExpression();
14:   partners = factory.getJoinPartners(rewriterSymbol);
15:   partners.add(rewriterSymbol);
16:   for all path ∈ partners do
17:      /* find the node in the subgoal tree representing the variable of the exported symbol */
18:      variable = path.getVariable();
19:      node = subgoal.getNode(variable);
20:      /* find or create inner nodes in the subgoal tree for receiver schema elements */
21:      for all name ∈ nodeNames do
22:         if subgoal.existsReceiverNode(name) then
23:            node = subgoal.getReceiverNode(name);
24:         else
25:            newNode = new Node(‘’, name);
26:            node.addNode(newNode);
27:            node = newNode;
28:         end if
29:      end for
30:      /* create inner nodes in the subgoal tree for rewriter schema elements, traversal from left to right */
31:      for all element ∈ path.getInnerNodeNames() do
32:         /* traversal from left to right */
33:         if subgoal.existsRewriterNode(element) then
34:            node = subgoal.getRewriterNode(element);
35:         else
36:            newNode = new Node(element, ‘’);
37:            node.addNode(newNode);
38:            node = newNode;
39:         end if
40:      end for
41:      /* create leaf node in the subgoal tree */
42:      leafName = path.getLeafName();
43:      leafNode = new Node(leafName, receiverNode.getName());
44:      leafNode.setExported();
45:      node.addNode(leafNode);
46:      node = leafNode;
47:      /* look for predicates */
48:      for all predicate ∈ factory.getPredicates(path) do
49:         node.setPred(predicate);
50:      end for
51:   end for
52: end if
```

both structures. This information can be obtained from the predicate factory, which has learned it through Algorithm 1. In the view definition of Figure 4.2 a leaf node in the receiver schema is `title`. It contains the exported symbol `$b/title`.

For the so found path (corresponding to the rewriter schema) and its join partners we need to create all nodes in the subgoal tree if they do not yet exist. For this purpose, the exported symbol (rewriter schema) is divided into three parts: the variable, the inner node path, and the leaf node, e.g., `$a/book/title` would be divided into (i) `$a`, (ii) `book`, and (iii) `title`. At first, we consider the variable (lines 17–29). We need to find the node within the subgoal tree that represents the variable, e.g., variable `$a` represents the path

`literature/author` such that the node we need to locate is the one corresponding to `author`. Such a node must exist since it has been created by Algorithm 1. As children of the so found node in the subgoal tree we create inner nodes – if not yet existing – for all receiver schema node names stored in *nodeNames* (first-in-first-out). Then, all inner nodes corresponding to the inner node path (e.g., `book`) are created (lines 30–40). Afterwards, a leaf node is created (lines 41–50), which contains both a receiver schema element name (i.e., the node in the receiver schema that contained the exported symbol) and a rewriter schema element name (i.e., the node in the rewriter schema referred to by the exported symbol) – in our example both are named `title`. The leaf node is marked as exported and if necessary assigned predicates (lines 48–50). In our example we obtain a leaf node in the subgoal tree of Figure 4.2 that contains the receiver node name `title` derived from `JRRTolkien/book/title` and the rewriter node name `title` derived from `$a/work/title`.

4.1.2 Creating Subgoal Trees for Queries

In order to compare a view to a query, queries have to be transformed into a subgoal tree representation as well. Such a tree is constructed in a similar manner. The main difference is that in case of query subgoal trees there is no receiver schema and thus there are no receiver nodes that need to be considered. Figures 4.3 and 4.4 illustrate the subgoal trees created for the two example queries that we have introduced in Section 3.3 and illustrated as POP trees in Figure 3.5.

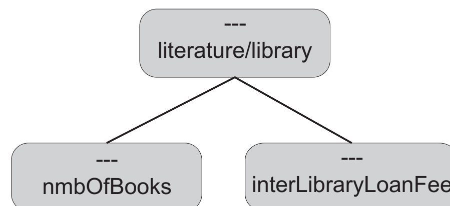


Figure 4.3: Subgoal Trees for the Example Skyline Query. *Exported nodes are tagged (grey).*

Algorithm 3 sketches the algorithm that creates subgoal trees given a query POP tree as input. It begins with identifying all select POPs on leaf level (line 1). For each of these select POPs (remember that a POP on leaf level is always a select POP) a separate subgoal tree is created since such a POP directly corresponds to a subgoal. In principle, we follow the path from the leaf POP to the root POP in the query tree and simultaneously construct the subgoal tree (lines 3–29). At each visited POP we consider its parameters (e.g., the path expressions used to define the ranking function of a topn POP), its predicates, and the exported data. So, we first extract the POP's parameter paths (line 4). If the currently considered POP is a leaf level select POP, we create and initialize a new subgoal tree (lines 6–9). The root node of this subgoal tree is assigned the full path of the select expression (line 8), e.g., `literature/library` in Figure 4.3. For non-leaf-level POPs we need to check whether the schema nodes referred to by the paths of the parameters are already represented within the subgoal tree by corresponding nodes. If necessary, such nodes are created and inserted into the subgoal

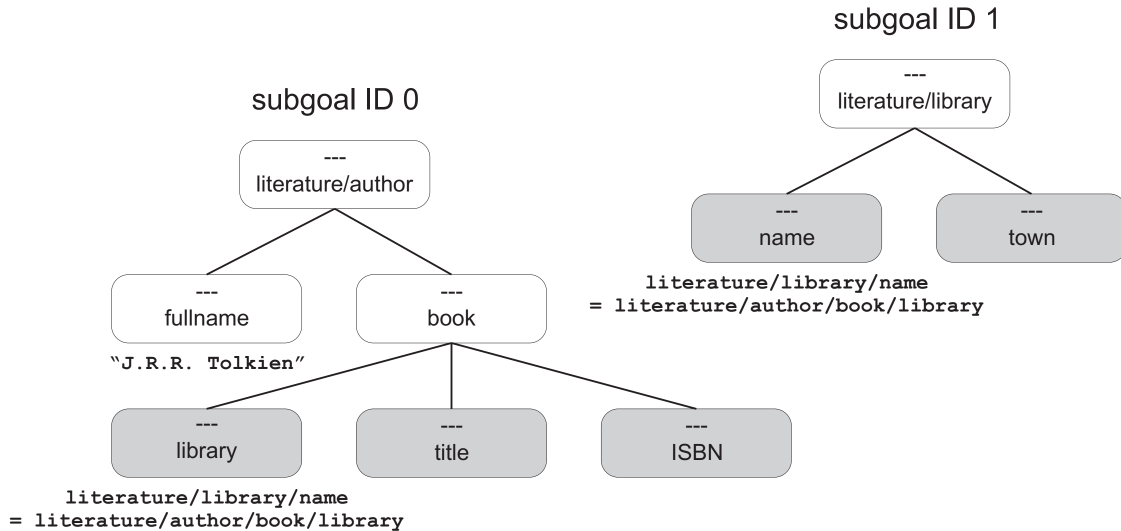


Figure 4.4: Subgoal Trees for the Example Join Query. *Exported nodes are tagged (grey).*

tree (lines 12–20). In our example skyline query – after having created the root node – the paths (`library/nmbOfBooks` and `library/interLibraryLoanFee`) referred to by the definition of the skyline POP's ranking functions entail the creation of the two nodes `nmbOfBooks` and `interLibraryLoanFee` in the subgoal tree.

After having created and found all nodes to represent the currently considered path expression, we need to decide whether to mark nodes of the path as exported or not (line 23). This depends on the type of the currently considered POP as well as on the query. In contrast to view subgoals, inner nodes of query subgoals might be marked as exported. This happens if the query does not use any construct or select POPs to specify the structure of the result with respect to the subgoal. In this case all nodes in the subgoal tree are marked as exported (see Figure 4.3 for an example). In general we need to consider what data is output as a result of a POP. Leaf nodes in the subgoal tree referred to by topn and skyline POPs are always marked as exported (assuming of course there is no other more restrictive POP above them in the query tree) because the data contained in these nodes is used for ranking the records. Join POPs as well as union POPs are considered neutral and do not cause any nodes to be marked as exported. In case of a select POP p, a leaf node n is only marked as exported if there exists no other construct or select POP on the path between p and the root node of the query POP tree that prunes the schema element represented by n. In general, construct POPs always result in marking the leaf nodes that their parameters refer to as exported. There is only one exception: the leaf node is not marked as being exported if the construct POP using it has been created due to rewriting and the original query had no construct POP. The reason is that when the construct POP is the result of the rewriting process, the user did not express his/her interest in those elements and thus might actually have no interest in some of the referenced schema nodes.

As for creating subgoal trees for views, predicates have to be treated in a special way (line 24). POPs may contain predicates in their XPath expressions. For example, `nmbOfBooks` $>$ `'10000'` could be a predicate for `literature/library` denoted as

Algorithm 3 *createQuerySubGoalsQuery(query)*

```
1: leaves = query.getLeafPOPs();                          /* find POPs on leaf level */
2: for all pop ∈ leaves do
3:   while pop is not null do
4:     parameterPaths = pop.getParameterPaths();    /* extract path expressions from the POP's parameters */
5:     for all path ∈ parameterPaths do
6:       if pop ∈ leaves then
7:         /* create new subgoal tree and root node */
8:         node = new Node(path);
9:         subgoalTrees.add(new Subgoal(node);
10:      else
11:        /* find/create a subgoal tree node for each element name in path, traversal from left to right */
12:        for all element ∈ path do
13:          if existsNodeInSubgoalTree(element,subgoalTrees) then
14:            node = getExistingNodeInSubgoalTree(element,subgoalTrees);
15:          else
16:            newNode = new Node('', element);
17:            node.add(newNode);
18:            node = newNode;
19:          end if
20:        end for
21:      end if
22:      /* mark subgoal tree node as exported if necessary */
23:      node.setExported(pop.checkParametersAreExported());
24:      predicate = pop.getPredicate();
25:      /* collect and set predicates */
26:      factory.createPredicatesInQuery(predicate);
27:    end for
28:    pop = pop.getParent();
29:  end while
30: end for
31: /* consider joins when all subgoal tree nodes have been created */
32: factory.treatJoinPredicatesForQuery();
33: return subgoalTrees;                                   /* return query subgoal trees */
```

`literature/library[nmbOfBooks>'1000']` in the XPath expression of a select POP. Again, we use a predicate factory (line 26) to collect and assign predicates to nodes in the query subgoal tree. Since predicates involved in join POPs also hold for the join partner, we again need to treat this case separately at the end once all predicates are known (line 32). The subgoal trees corresponding to our example join query are illustrated in Figure 4.4. In order to distinguish the created subgoal trees, each of them is given a subgoal ID, in the example "0" for the left and "1" for the right subgoal tree.

4.2 Query Rewriting Using Subgoal Trees

The algorithm we present in this section works in a completely decentralized manner and pays attention to rank-aware query operators using the Bucket Algorithm (Section 2.6.3) as basis. Given a query (formulated on a peer's local schema), a set of neighbors, and corresponding mappings, the algorithm generates a query plan that contains rewritings combined by union POPs. Each rewriting is likely to contain *remote queries* representing subqueries to be forwarded to neighboring peers. These subqueries are formulated on the receiver's local schema and contain construct POPs allowing the receiver to transform its results into the schema of the sender so that the sender always receives results in its local schema. In principle, the rewriting algorithm works in nine steps:

1. receiving a query plan (POP tree)
2. preprocessing
3. creating buckets
4. sorting view subgoals into buckets
5. creating combinations of buckets
6. creating query snippets
7. creating initial rewritings
8. optimizing remote queries and rewritings
9. assembling the rewriting result (rewritten query plan)

We illustrate these steps using the two running examples introduced in Section 3.3: the skyline query of Figure 3.5(a) and the join query of Figure 3.5(b). Both queries are issued at P_0 in our example network of Figure 3.2 and hence formulated on P_0's local schema.

4.2.1 Receiving a Query Plan

When the query is issued, the first step for P_0 is to evaluate the query on its local data. As we do not distinguish between the schema a peer makes accessible to other peers in the network (peer schema) and the schema the data is actually stored in locally but assume both are the same, no rewriting is necessary to answer the query locally. Thus, the query plan that is given as input to the rewriting algorithm may already contain intermediate results originating from local query processing – the local result of a skyline computation is for example attached to the skyline POP. Depending on the strategy and query operators (Chapter 6), it might be worthwhile to forward such intermediate results along with the query to neighboring peers. These peers can use this information to prune additional peers from consideration [92]. In principle, input query plans correspond to queries formulated in L_Q according to Definition 3.3.1 without remote query POPs. There are only two more requirements that have to be fulfilled in order to apply our rewriting algorithm:

1. If a query contains multiple joins, these joins have to be directly connected to each other, i.e., no other POPs are allowed on the path between any two joins in the query tree (step 8, Section 4.2.8).
2. If a query contains both joins and rank-aware operators, the join must be located in the POP tree directly underneath the rank-aware operators or the rank-aware operators must be direct children of the *join* POP (step 2, Section 4.2.2).

Note that these two requirements do not restrict the space of possible queries but only influence the structure of a query plan tree – plans can be reorganized accordingly before handing them to the rewriting component.

In general, the input POP tree is assigned a set of “relevant neighbors” that are to be considered for rewriting. This set has been determined by data-level pruning (Chapter 5) and is likely to contain fewer neighbors than the peer is connected to. As rewriting costs strongly depend on the number of considered views, this information significantly improves performance. However, for our two example queries we assume that none of the neighbors has been pruned on data-level.

4.2.2 Preprocessing

If a query has been received from a neighboring peer, it has a remote query POP as root node of the query tree. As this node only encodes the information to which peer the result of the query is to be sent to, it is ignored by the rewriting component and not considered in the following.

Both union POPs and rank-aware POPs need to be treated in a special way if we want to use the Bucket Algorithm as basis. Queries containing union POPs can always be split up into several subqueries, which can be processed separately and whose results simply need to be merged at the peer that rewrites the query. Thus, before rewriting a query that contains unions, subqueries not containing union POPs are extracted. In order to distinguish between the subqueries, they are given unique identifiers. After having rewritten the subqueries independently from each other, we assemble the rewriting result for the original query. For this purpose, we use the original query and replace the subtrees corresponding to the subqueries with their rewriting results. The resulting query may now be processed to answer the query. This kind of preprocessing is illustrated in Figure 4.5.

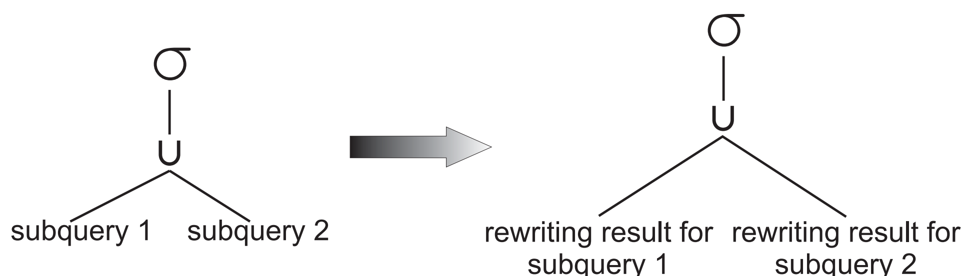


Figure 4.5: Preprocessing of a Query with Union POPs. *Left: Original Query, Right: Rewritten Query*

Rank-aware operators are not part of conjunctive queries either. With respect to queries containing POPs such as topn and skyline, we basically proceed the same: we split up such queries into several subqueries such that each can be treated as an individual query without such operators. Assume we have a two-level POP tree with a skyline POP as root and a select POP underneath (our example skyline query of Figure 4.3). Simply rewriting the subquery underneath the skyline POP results in a situation in which the initiating peer receives all data records from its neighbors corresponding to the select expression. Doing so, the initiator has to bear all the computational load of computing the skyline. Furthermore, network load would be enormous since all data records of possibly all neighbors would be rewritten and sent to the initiator. Obviously, this is not an efficient solution, especially when we consider that most of the effort is unnecessary because only a small portion of the data is actually needed to answer the query.

An effective solution is sharing the information about the skyline operator with the neighbors, i.e., considering the operator for rewriting. The decisive characteristic of rank-aware operators, which we can use for this purpose, is their additivity that results from the additivity of the aggregate functions they are defined on (MIN, MAX):

Observation 4.2.1 (Additivity of Skyline and Top-N Operators)**.** *Let ϕ denote a skyline or a top-N operator and let $D_1, \ldots, D_n$ denote data sets that ϕ can be evaluated on. Then, the following equation holds for all ϕ:*

$$\phi(D_1, \ldots, D_n) = \phi(\phi(D_1), \ldots, \phi(D_n))$$

Thus, evaluating ϕ on the union of the individual data sets is the same as evaluating ϕ on the data sets in separate and afterwards on the union of their results. Hence, when neighbor peers receive a rewritten query that includes ϕ, network load as well as computational load at the initiator can be reduced by having the neighbors evaluate ϕ. Once the initiator has received the results from its neighbors, it once more evaluates ϕ over the union of its local result and the results received from its neighbors.

Whenever a query is given as input that contains rank-aware operators, the preprocessing step takes care of introducing such possibilities by cloning the operator and inserting it on top of the rewriting result. More formally, we locate the bottommost rank-aware operator and use all operators underneath for rewriting. The operator is cloned and inserted on top of the rewriting result. The part of the original query above the operator is computed locally at the peer that rewrites the query using the results provided by the rewritings as input.

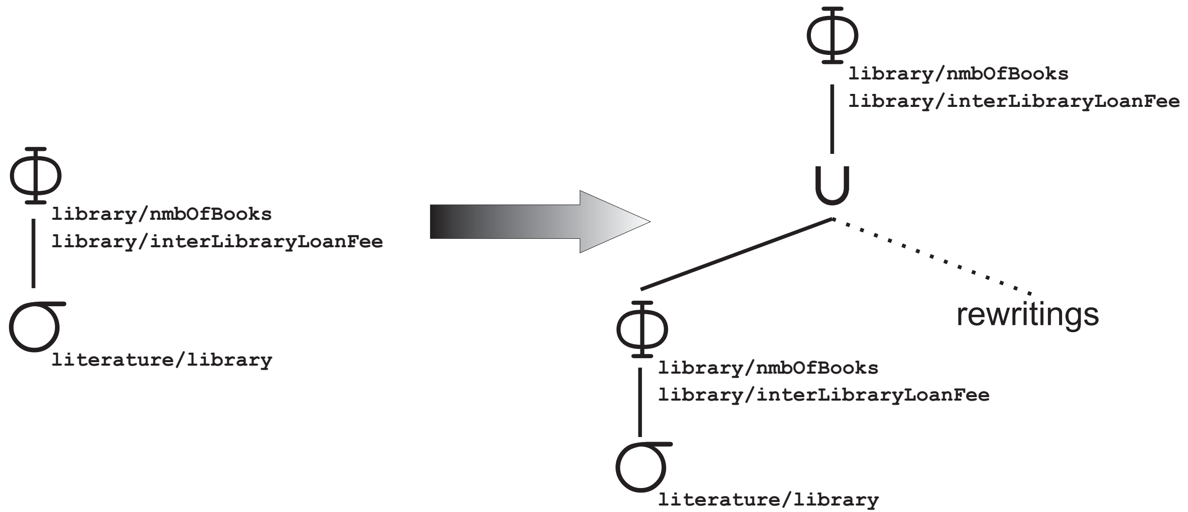


Figure 4.6: Preprocessing of a Skyline Query

Let us again consider our example skyline query of Figure 4.3. The query POP tree corresponding to the rewriting result is sketched in Figure 4.6 – the original skyline POP of the query is cloned and inserted as root node of the rewriting result. The union POP underneath defines that the result of the local query (left child), i.e., the original input query evaluated locally, is merged with the results of the neighbors (right child, dotted line), i.e., the query rewritings. As this is only the preprocessing step, the right child of the union POP is not yet known but computed in the following steps of the rewriting process.

The considerations above only hold for rank-aware queries that do not contain join POPs. Assuming we have a query that contains a skyline POP underneath a join POP (Figure 4.7). In the preprocessing step we clone the skyline operator and insert it on

top of the join POP – in case there are multiple join POPs the clone is inserted on top of the topmost join POP. For such queries we can consider the join POPs as well as all operators underneath for rewriting. The reason is that joins do not make any projections, i.e., it is not possible to "lose" any data that might be relevant to evaluate the rank-aware operator.

Figure 4.7: Preprocessing of a Skyline Query with Join

These considerations led to the requirement (step 1, Section 4.2.1) that in order to consider queries containing both joins and rank-aware operators for rewriting, there must not be any other POP that might perform any kind of projection or transformation of the data in between join and rank-aware operator. However, it might make a difference whether a rank-aware operator is located above the join or underneath.

We now consider the examples illustrated in Figures 4.7 and 4.8. These figures illustrate the preprocessing step for queries containing join and skyline POPs. Although both queries are very similar, the result sets are not necessarily the same. It is only the same if `literature/library/name` and `literature/author/book/library` are foreign keys, i.e., for each library there exists at least one `book` element that fulfills the join condition. The considerations that lead to this observation are as follows.

Assuming there was a library l in `literature/library` that had no join partner b in `literature/author/book`. We further assume l is part of the skyline on libraries directly computed on the output of the select operator (Figure 4.7). Therefore, l might dominate and thus prune a set of other libraries $\mathcal{L}$ – see Section 2.1.2 and Definition 2.1.3 for details on skyline queries and the dominance relationship. After having processed the

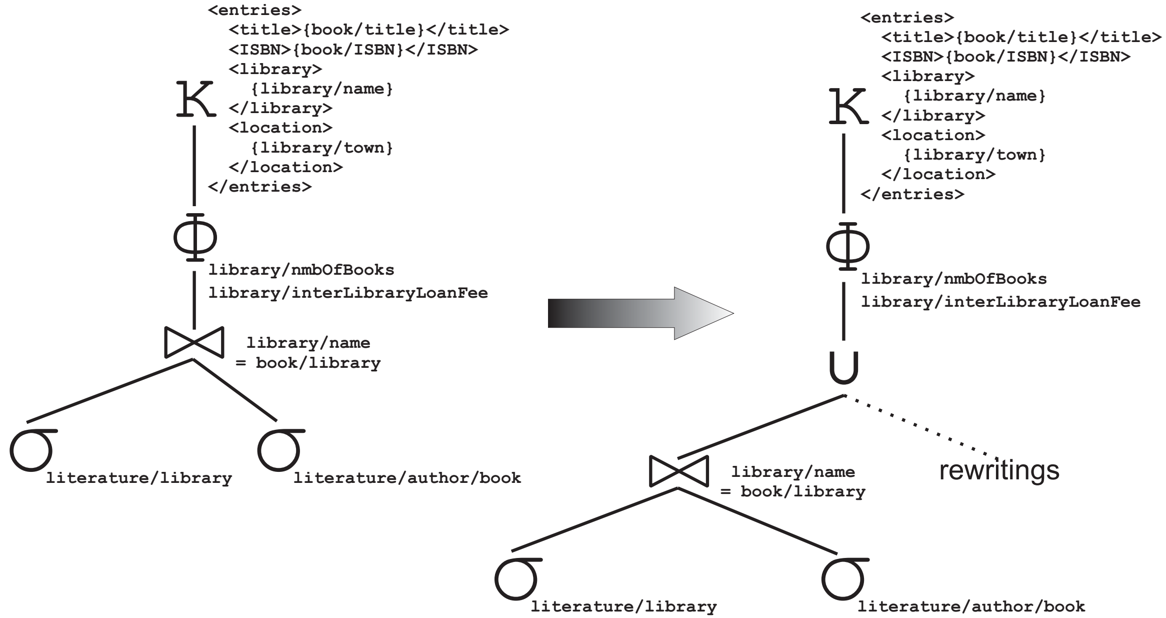


Figure 4.8: Preprocessing of a Skyline Query with Join – Alternative

join POP, l would not have any representative l' in the join result and thus would not be part of the result. In the query of Figure 4.8, where the skyline is processed after the join, there is no l or l' that could dominate any library in $\mathcal{L}$. Thus, libraries in $\mathcal{L}$ could still be part of the skyline. To guarantee the same result (no foreign key relationship) in the query of Figure 4.7, in which the join is processed on top of the skyline result, we would have to reconsider all libraries in $\mathcal{L}$ as they now, after the removal of l, could actually be part of the skyline. The same considerations apply to the top-N operator in a similar fashion. The query parser can be used to solve this problem by automatically reordering the operators such that the join POP is always underneath the rank-aware operator. As in this case some computational load is shifted to the peer that issues the query, the user can explicitly configure the parser to arrange rank-aware POPs above or underneath join POPs.

With respect to the preprocessing step, future work might also consider optimizations for the problem of rank-aware query operators in conjunction with joins. For instance, so far we only locate the bottommost rank-aware operator and use it with all operators underneath for query rewriting. It might be possible to optimize our algorithm so that it not only considers all operators underneath but also the operators above.

4.2.3 Creating Buckets

In this step we identify the subgoals that are contained in the query. For this purpose we use the algorithm introduced in Section 4.1.2 and obtain one subgoal tree for each subgoal. For each subgoal one bucket is created. Figures 4.3 and Figure 4.4 illustrate the subgoal trees we obtain for the two running example queries.

We add some additional information to each bucket. First, each bucket is assigned a subgoal ID indicating which of the query's subgoals it belongs to. In the following, we

often refer to this subgoal ID as bucket ID as it also unambiguously identifies a bucket. Furthermore, a bucket knows the total number of subgoals created for the query and the query ID. This information will be necessary to create remote query POPs for query snippets in step 6 (Section 4.2.6).

4.2.4 Sorting View Subgoals into Buckets

The goal of this step is to identify view subgoals that might be relevant to answer the query and to sort them into "matching" buckets. For each peer contained in the list of relevant neighbors the corresponding view subgoal trees are created according to Section 4.1.1 and compared to the query subgoals of the buckets. If a view subgoal is able to answer a query subgoal, i.e., if the query subgoal is contained in the view subgoal, the view subgoal is sorted into the corresponding bucket.

We have already pointed out in Section 2.6.2 (Definition 2.6.3) that finding a symbol mapping from the query to the view is a sufficient condition to decide on containment. This is the case if a variable mapping (element mapping) between query subgoal and view subgoal can be found. Intuitively, if the following conditions are fulfilled, then a mapping exists:

1. All paths and element names that occur in the query subgoal tree need to have a counterpart in the view subgoal tree.
2. Exported leaf nodes in the query subgoal tree must correspond to exported leaf nodes in the view subgoal tree or to leaf nodes with constants.
3. Predicates in the query subgoal tree must be satisfiable in the view subgoal tree because a query subgoal that cannot be fulfilled by the view subgoal would lead to an empty answer.
4. Since for rewriting the input query the view subgoals have to be combined, the view has to contain either the same join predicate or the view subgoal needs to have exported nodes that the rewriter element involved in the query's join can be mapped to.

Deciding on containment of complex objects and XML trees [51, 125, 137, 187] is similar to the problem of deciding on containment of subgoal trees. As subgoal trees basically represent XML trees, the definitions of XML tree embedding and containment proposed in [51, 187] are very similar to our definition of subgoal tree containment and containment mappings (Definition 4.2.2).

Definition 4.2.2 (Subgoal Tree Containment Mapping)**.** *A containment mapping from a query subgoal tree T_Q to a view subgoal tree T_V is a node mapping ($\psi : \mathcal{N}(T_Q) \mapsto \mathcal{N}(T_V)$) such that:*

- *ψ maps the root node r_Q of T_Q to a node $r_V \in \mathcal{N}(T_V)$,*
- *if node n'_Q is a child node of $n_Q \in T_Q$, then $\psi(n'_Q)$ is a child node of $\psi(n_Q)$,*
- *the rewriter path expression defined by the labels of nodes on the path from r_Q to a node $n_Q \in T_Q$ corresponds to the rewriter path expression defined by nodes on the path from r_V to $\psi(n_Q)$,*
- *ψ maps each exported leaf node in T_Q to an exported leaf node in T_V or to a leaf node with a constant in T_V, i.e., $\forall e \in \mathcal{E}(T_Q) : \psi(e) \in \mathcal{E}(T_V) \vee \psi(e) \in \mathcal{C}(T_V)$,*

- *predicates assigned to a node* $n_Q \in \mathcal{N}(T_Q)$ *imply predicates assigned to* $\psi(n_Q)$ *so that* $\mathcal{P}(T_Q) \Longrightarrow \mathcal{P}(T_V)$*, and*
- ψ *maps a node* $n_Q \in \mathcal{N}(T_Q)$ *with an assigned join predicate either to a node* $\psi(n_Q) \in \mathcal{E}(T_V)$ *or to a node* $\psi(n_Q) \in \mathcal{N}(T_V)$ *with the same join predicate.*

The fact that nodes in a view subgoal tree already encode mappings between rewriter and receiver schema elements, makes finding a symbol mapping straightforward and comes down to the problem of finding matches in the view subgoal tree for all nodes in the query subgoal tree. This means all that needs to be done is recursively traversing the view subgoal tree T_V in correspondence to its rewriter schema element annotations and the rewriter schema elements represented by nodes in T_Q.

At first, we need to find the node in T_V corresponding to T_Q's root node. This is straightforward as only the path expression of T_Q's root node has to be compared to the rewriter path expression of T_V's root node. If the latter represents only a subpath, we traverse the tree until we find the corresponding node. For this process intermediate nodes exclusively representing nodes in the receiver schema are ignored. For all other nodes in T_Q we basically proceed the same to find their matches. If one of the conditions stated above (Definition 4.2.2) is not fulfilled, we stop the matching process and do not sort the view subgoal T_V into the bucket corresponding to T_Q.

Just like MiniCon [162] and unlike the Bucket Algorithm [84] we pay attention to predicates at this early stage of the rewriting process in order to avoid a large-scale containment test for rewritings of views that obviously do not yield any results. In contrast to the Bucket Algorithm not views are sorted into buckets but view subgoals.

In a view subgoal tree that is inserted into a bucket all its nodes matching nodes in the corresponding query subgoal tree are tagged so that non-tagged nodes do not have to be considered in the following. A special tag is assigned to the node matching the context node of the query subgoal tree (important for step 6, Section 4.2.6). Predicates, e.g., join predicates, assigned to nodes in the query subgoal tree are also assigned to their matches in the view subgoal tree (step 5, Section 4.2.5). Figure 4.9 shows a tagged view subgoal tree corresponding to the mapping from P_0 to P_4, which is inserted into bucket 1 of the join query. The view itself is assigned a list of bucket IDs that at least one of its subgoals has been sorted into. This list will be useful for reducing the number of rewritings that need to be considered in step 5 (Section 4.2.5).

4.2.5 Creating Combinations of Buckets

The buckets and the view subgoal trees sorted into them only represent parts of the query. The view subgoals within the buckets have to be combined into rewritings in order to answer the query. In a first step, this is done by building the Cartesian product of the view subgoals of all buckets – each resulting rewriting comprises exactly one subgoal from each bucket.[1] This Cartesian product may be rather large and may as well contain a lot of redundant combinations. To remove redundant combinations, the bucket IDs that have been assigned to the views in step 4 are consulted. These bucket IDs refer

[1] If there is an empty bucket, the rewriting process ends at this point because there exists no combination of view subgoals, i.e., no rewriting, that could answer the query.

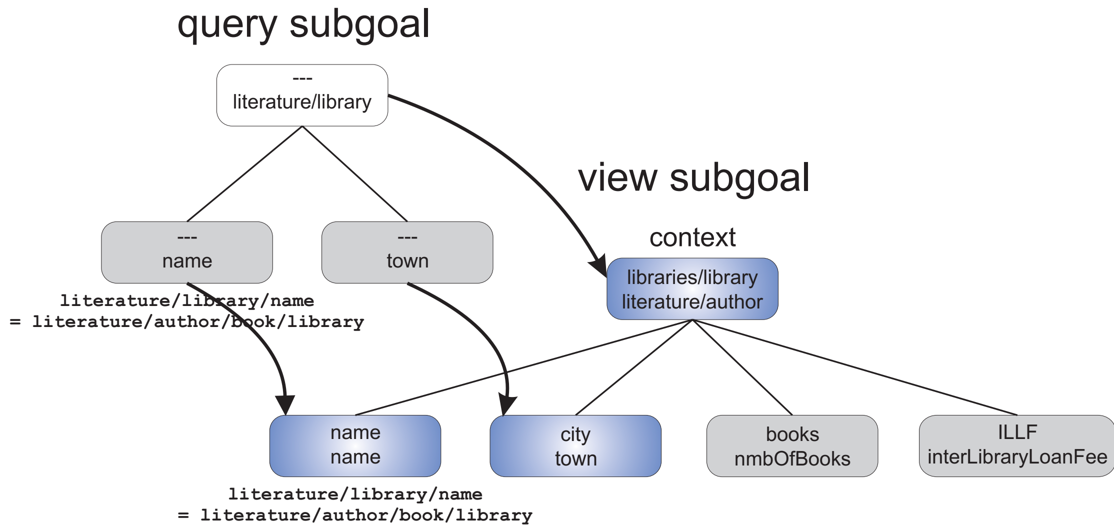


Figure 4.9: Sorting a View Subgoal of P_4 into Bucket 1 of the Example Join Query and Tagging Matching Elements (blue)

to all buckets which any subgoal of the corresponding view has been sorted into. All combinations that do not fulfill the following requirement are pruned:

$$\forall x \neq y \ \wedge x \textit{ is combined in a rewriting with } y : Colors(x) \cap Colors(y) = \emptyset \qquad (4.1)$$

x and y are views and $Colors(x)$ denotes the set of query subgoals (bucket IDs) that view x fulfills. In case a view fulfills multiple query subgoals, it has been given multiple bucket IDs. By checking the requirement above, the minimal set of additional views is found to answer the whole query. All combinations that do not fulfill the requirement can be pruned. This step works in a similar fashion as the MiniCon algorithm [162].

For the remaining combinations the algorithm checks whether the predicates of combined subgoals originating from different views are contradictory. In order to consider also predicates defined in the query, we have annotated nodes in the view subgoals with predicates of their “matching” nodes in the query subgoals in step 4 (Section 4.2.4). Since a combination of subgoals with contradictory predicates would yield empty result sets, such combinations are removed.

For our join query two buckets have been created (Figure 4.4). View subgoals of P_1 and P_2 have been sorted into bucket 0 and view subgoals of P_2 and P_4 have been sorted into bucket 1. The Cartesian product results in four combinations: (1,2),(1,4),(2,2), and (2,4). But as the view of P_2 is able to answer the whole query (i.e., both query subgoals), all combinations of P_2's view and other views can be removed. In the end only the combinations (1,4) and (2,2) remain. Note that the buckets contain view subgoals not views. Thus, the combination (2,2) denotes that two subgoals both supported by P_2 need to be combined.

Although there are many situations which allow for pruning combinations of subgoals in the way described above, there are some for which this kind of pruning may result in a loss of completeness. We might lose answers whenever it is possible that the pruned combination of any two subgoals yields records we do not obtain otherwise. An example is the use of “null” values. Assuming P_2 holds a null value for a library's town/city attribute

whereas P_4 provides the correct value. In this case we should not prune combination (2,4) although it can be considered redundant from a schema-level point of view. A similar situation arises when P_2 and P_4 provide different values for the town/city attributes. Without knowing which one is correct in advance, it might be worthwhile to retrieve both. In summary, for our example join query this means that although P_2 answers both subgoals, we should not prune the combinations (1,2) and (2,4) if we want to obtain the maximum set of result records and we cannot be sure whether P_1 or P_4 provide any join partners with data not stored locally at P_2. Moreover, in such situations it is not enough to consider all combinations of the views, we also need to consider combinations with subgoals of the rewriting peer's local schema. As the query is formulated on the schema of the rewriting peer, the algorithm can sort subgoals corresponding to the local schema into each bucket. These subgoals have to be considered when building the Cartesian product.

Obviously, this procedure, although it guarantees to compute the maximum set of result records, is not efficient in a PDMS because of the large number of combinations and the huge effort this entails for query processing. In addition, it is likely to produce duplicates that should be identified and removed by applying duplicate detection [87,199] or maybe additionally data fusion techniques [173, 206]. However, if we assume the data provided by the peers to be consistent and complete in the sense that attributes which can be used to formulate a join can be considered foreign keys in the peer's local database and there are no join partners held by other peers with data that is not stored locally, we can still perform the pruning as introduced above. We have to be aware that in some cases by applying this kind of pruning we run the risk of missing some result records but in consideration of the expected reduction in execution costs, we argue that taking the risk is an acceptable solution. Thus, for our example join query we prune the combinations (2,4) and (1,2) and consider only (1,4) and (2,2) in the following steps.

4.2.6 Creating Query Snippets

In the previous steps we have checked whether a view subgoal is useful for rewriting the query (step 4, Section 4.2.6) and whether a combination of view subgoals would possibly yield new and non-empty result sets (step 5, Section 4.2.6). In this step, we create remote queries based on the tagged view subgoal trees. For each view subgoal tree in a combination (output of step 5) we create one *query snippet*. A query snippet is a linear POP tree that has the following base structure: a remote query POP as root node, a construct POP as child of the remote query POP, and a select POP as child of the construct POP.

The snippet POP tree is created in a bottom-up fashion. Thus, we begin with creating the select POP on leaf level. For this purpose, we start at the root node of the view subgoal tree and traverse it downwards along tagged nodes until we find a node with multiple tagged child nodes. The path from the root node to the one that we stopped at determines the expression of the select POP, which may be extended by additional predicates (assigned to the tagged nodes) and/or constraints (assigned to the view subgoal).

The next step is to create the construct POP. Based on the node in the view subgoal tree that the query context node has been mapped to (this node has been marked in

step 4, Section 4.2.4), the expression of the construct POP is generated. The construct expression is built using the rewriter schema as basis. As the view subgoal tree contains the schema elements of both rewriter and receiver, we can obtain all necessary information from the view subgoal. We only need to consider tagged elements because all others have been identified as being unimportant to the query. At first, we extract the rewriter XML structure of the tagged nodes (considering only the subtree with the context node as root). For the tagged view subgoal tree of Figure 4.9 we obtain the following base structure:

```
<author>
  <name></name>
  <town></name>
</author>
```

The nodes of the view subgoal tree themselves indicate the receiver schema elements that the rewriter schema elements have to be mapped to. We obtain:

```
<author>
  <name>{ library/name }</name>
  <town>{ library/city }</name>
</author>
```

Finally, the remote query POP is created by exploiting the additional information we have assigned to the buckets in step 3 (Section 4.2.3). in order to obtain this information, we simply need to look it up in the bucket that the currently considered view subgoal has been sorted into. The remote query POP contains the ID of the neighbor whose view the view subgoal and thus the query snippet belongs to, the query subgoal ID (= bucket ID of the bucket the view subgoal has been sorted into), the number of subgoals the query contains, and the query ID (in case the original query has been partitioned into multiple subqueries). This information will become important later when answers from neighboring peers have to be inserted into the correct position in the query plan that is obtained as the result of the rewriting process.

The result of this step is a rewritten linear query POP tree, i.e., a query snippet, for each view subgoal involved in any combination. Note that if a view subgoal that has been sorted into a particular bucket is used in multiple combinations, we only need to create the query snippet once and replicate it for all other combinations. Figure 4.10 shows the query snippet corresponding to the tagged view subgoal of P_4, which has been inserted into bucket 1 of the join query (Figure 4.9).

4.2.7 Creating Initial Rewritings

The query snippets created in the previous step are remote queries performing selection, projection, nesting, unnesting, and renaming of elements. In principle, it is already possible to use them to generate rewritings as each query snippet corresponds to a select POP on leaf level in the original query. Thus, we can already generate a rewriting for each combination found in step 5 (Section 4.2.5) by using the original query as template for a rewriting and replacing the select POPs with the corresponding query snippets. By connecting the rewritings for all combinations with union POPs, we can obtain the rewriting result.

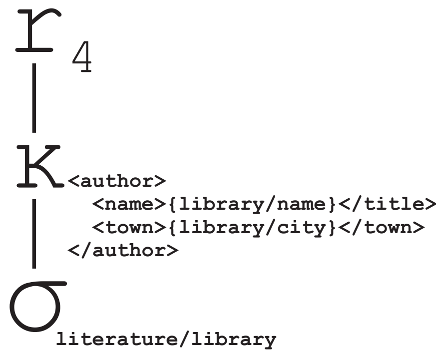


Figure 4.10: Creating Query Snippets from the Tagged Subgoal Tree of P_4 (Join Query)

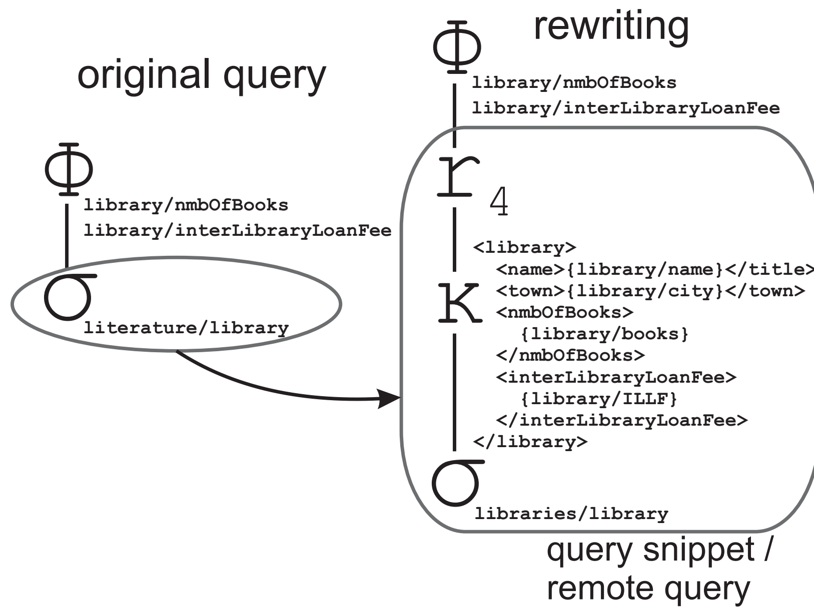


Figure 4.11: Generating Rewritings for the Example Skyline Query and a Query Snippet Originating from P_4

Hence, we begin the generation of rewritings for each combination (output of step 5) using the original query as template and replacing the select POPs on leaf level with the corresponding query snippets. Figure 4.11 shows such a rewriting for the example skyline query and a query snippet corresponding to the subgoal of P_4. In the following, we use the term *remote query* to denote subtrees with remote query POPs as root node since they are meant to be sent to neighboring peers and processed there.

Figure 4.12 illustrates the two rewritings for the example join query corresponding to the combinations (2,2) and (1,4). In order to create rewritings for queries involving joins, we do not use the complete original query as template but only the subtree with the join as root node. The reason is that all operators above the join POP are computed at the peer rewriting the query because the remote queries are located underneath the join. Thus, the operators above the join are not considered until step 9 (Section 4.2.9), which assembles the rewriting result for the input query.

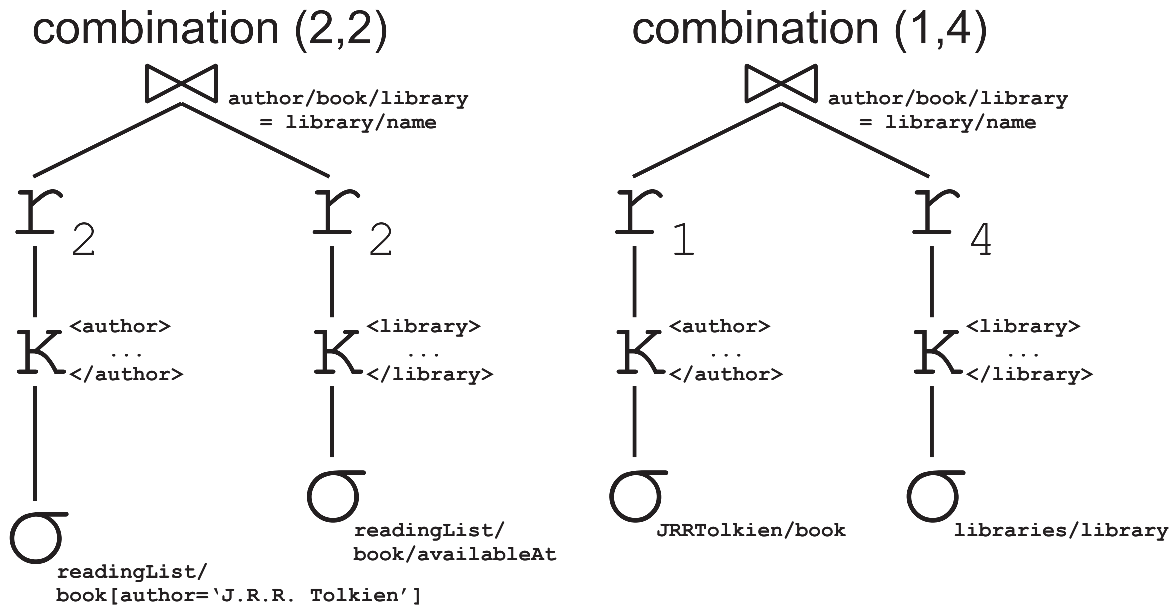


Figure 4.12: Generating Rewritings for the Example Join Query

4.2.8 Optimizing Remote Queries and Rewritings

After having generated these initial rewritings, we need to optimize them. The goal of the optimization is to push as many operators (not only rank-aware operators) of the original query POP tree down underneath the remote query POP and the construct POP of the remote query. The expression of the construct POP provides all information that is necessary to rewrite the path expressions contained in the parameters of such POPs. If necessary, even records representing an intermediate local result can be rewritten using the same technique such that the receiver can use those results for query processing (Chapter 6).

In our example skyline query rewriting of Figure 4.11, the skyline POP can be pushed down underneath the remote query POP and the construct POP such that the receiving peer can already evaluate the skyline operator on its local data and thus reduce the size of the result set. The result is shown in Figure 4.13 illustrating how the construct POP's expression is used to rewrite the ranking functions of the skyline POP.

Under certain circumstances it is possible to combine the remote queries of a rewriting, which has been generated in the previous step, into just one remote query. Whenever two remote queries of the same neighbor are "connected" via joins in a rewriting (the connection might involve multiple joins), we reorganize the tree such that the remote query POP can be pushed up above the join POP. Note that only a join POP might lead to such rewritings – this is the reason why we have required (Section 4.2.1) that when a query contains multiple joins, they must be directly connected to each other.

Combining two remote queries means that the join connecting them in the initial rewriting is computed not by the rewriting peer but already at the neighbor that holds the data. The resulting remote query contains the join POP with two children, each of which containing one construct POP and one select POP. However, this is still subop-

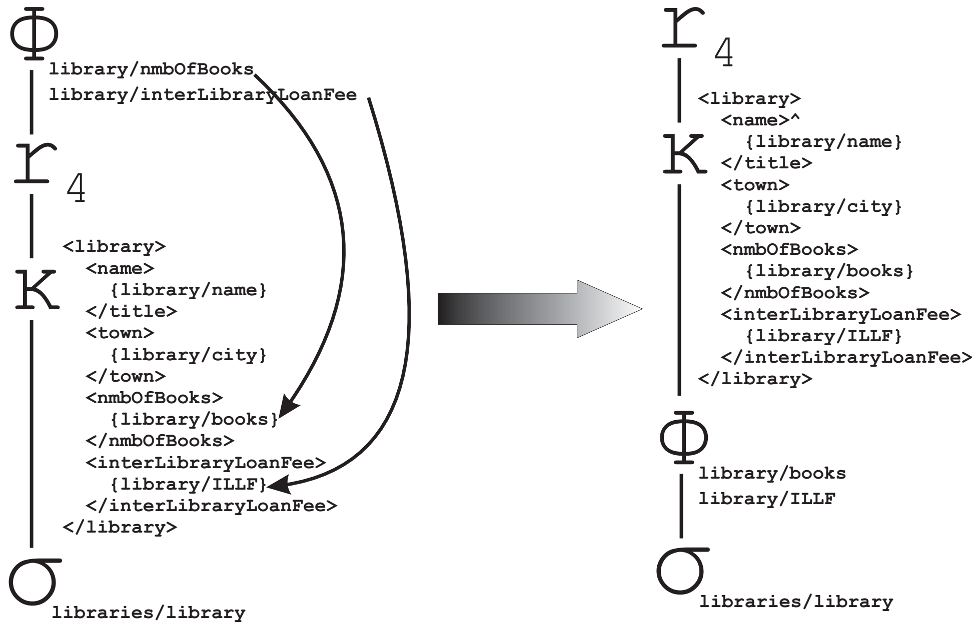


Figure 4.13: Optimizing Rewritings for the Example Skyline Query

timal because before evaluating the join, all records matching the two select POPs are transformed by the construct POPs. As many of them will be pruned by evaluating the join, it is worthwhile to combine the two construct POPs, adapt their construct expressions accordingly, and push the resulting construct POP up above the join POP. We can use the construct POPs' expressions to rewrite the paths referred to by the join POP's join predicate. Figure 4.14 illustrates this principle with the example of the join query's rewriting corresponding to the combination (2,2) of Figure 4.12.

If the view definition to the neighbor that the resulting remote query is to be sent to contains the same join as the query, then the data is already stored in the joined format at the neighbor. Thus, we can remove the join. This is illustrated in Figure 4.15 for the join query example and the combination (2,2) based on Figure 4.14.

For queries that contain multiple joins it might be necessary to reorder the joins so that remote queries corresponding to the same neighbor can be combined. At first, bushy trees are reordered into right oriented trees as illustrated in Figure 4.16. Afterwards, the algorithm recursively tries to reorder the joins such that remote queries corresponding to the same neighbor are children of the same join POP. Only then can we combine the two remote queries into one. The principle is sketched in Figure 4.17. Note that as this technique is applied recursively, in the end more than two remote queries might be combined into one.

When we sorted the view subgoals into buckets in step 4 (Section 4.2.4), we already checked whether a join predicate contained in the query is also contained in the view definition or if there are at least exported nodes in the view subgoal trees that can be mapped to the rewriter schema elements involved in the query's join definition. Consequently, step 5 (Section 4.2.5) in general only creates combinations of view subgoal trees that provide results. There is only one exception. Assume both query and view have only one join, then they have two subgoals each. Further assume the query's join predicate

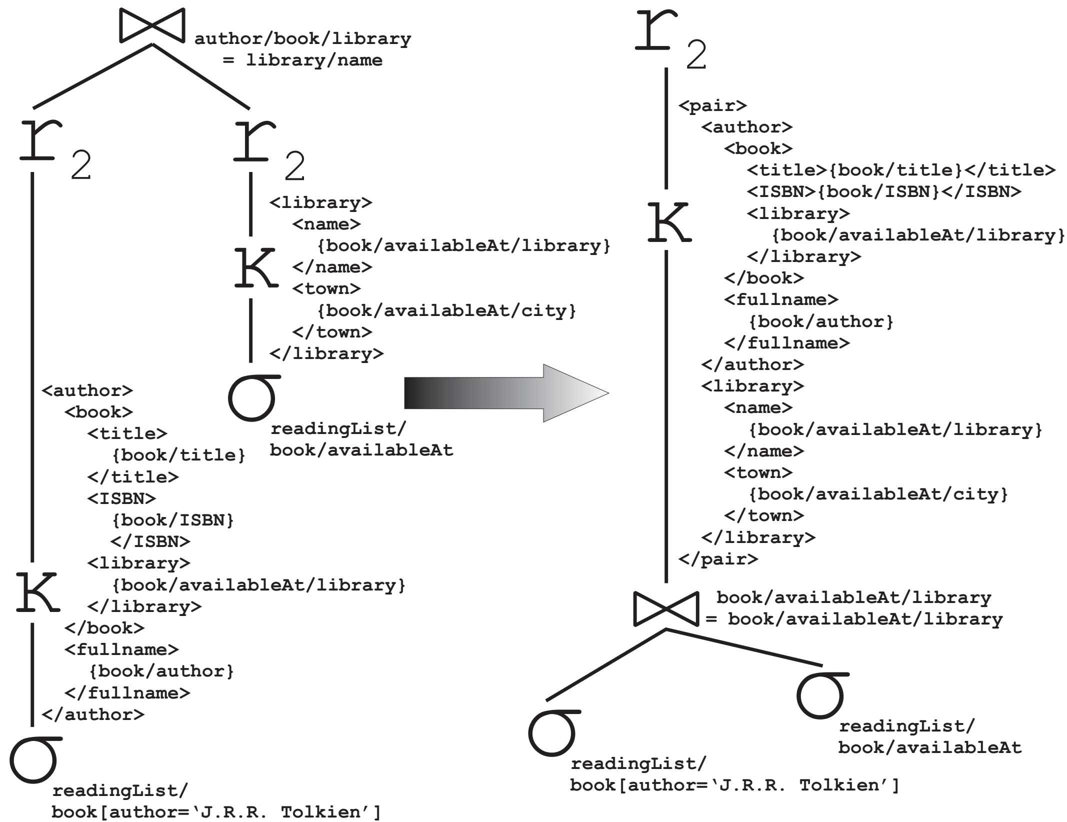


Figure 4.14: Combining Remote Queries of a Rewriting (Join Query)

is contained in the view. In that case we did not require the nodes in the view subgoal tree – representing the elements used in the join predicate – to be exported nodes (Section 4.2.4). It might be possible that only one of the two view subgoals fulfilled all the other conditions. Thus, only one of the two view subgoals would have been sorted into a bucket. This view subgoal might have been combined with a view subgoal originating from another mapping – resulting in a combination that should be pruned. Remember that if both view subgoals had been inserted, step 5 (Section 4.2.5) could have taken care of pruning all combinations of the two view subgoals with view subgoals originating from other mapping definitions.

To prune such combinations, we do an extra check on each rewriting that contains a join involving two view subgoals originating from different views, i.e., two remote queries corresponding to different neighbors. More precisely, we check whether the schema elements that are necessary to perform the join are contained in both remote queries. All we need to check is whether the two construct expressions of the construct POPs contain an expression for each schema element that the join is defined on. Only if this condition is not fulfilled, we additionally have to prune the corresponding rewriting, i.e., the corresponding view subgoal combination. For our example queries there are no such combinations.

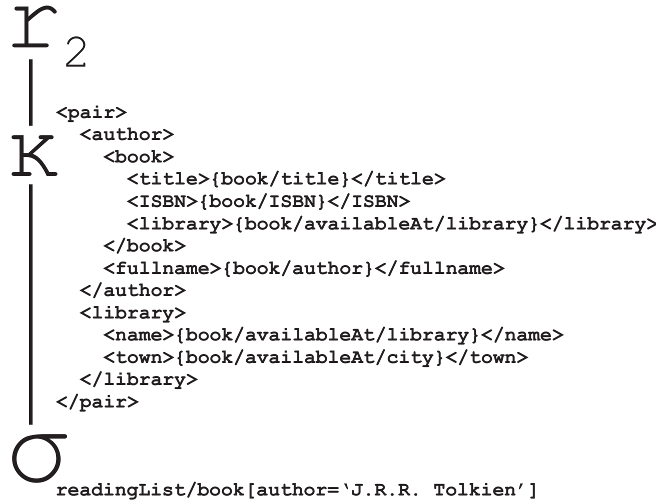


Figure 4.15: Removing a Join in a Rewriting when the Data is Stored in the Joined Format at the Receiver (Join Query)

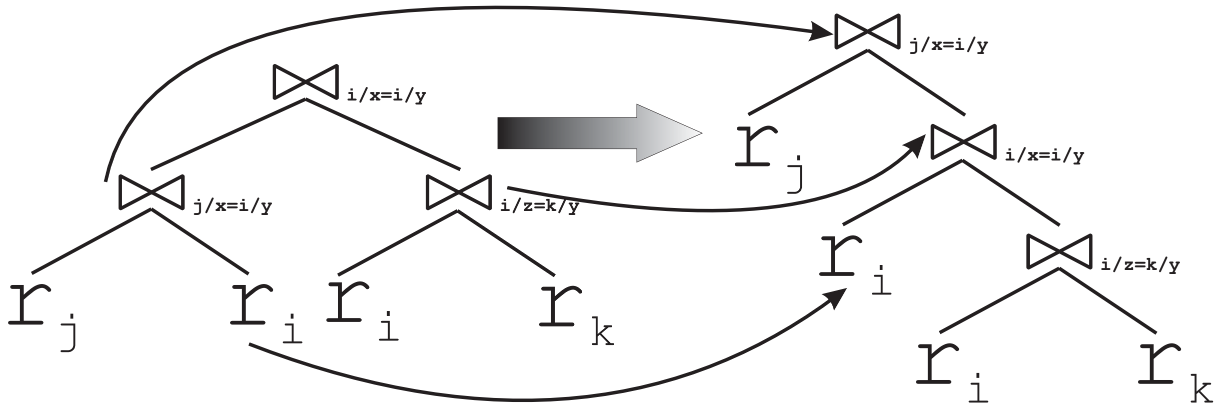


Figure 4.16: Reordering Joins in a Query

4.2.9 Assembling the Rewritten Query Plan

In the last two steps rewritings have been created and optimized. In this step the complete query plan, which corresponds to the result of the rewriting process, is built by combining the original query and all rewritings by means of union POPs. Figure 4.18 shows the resulting final query plans as the rewriting result for our two example queries.

In order to assemble the rewriting result, we need to consider the preparations we have made in the preprocessing step (Section 4.2.2). For the join query there were no preparations, thus assembling the final query plan comes down to connecting the rewritings with union POPs (Figure 4.18(a)). However, for the skyline query we made the preparations illustrated in Figure 4.6. These preparations sketched the final rewriting result and introduced a skyline operator as its root node. The final output of the rewriting process is created based on the plan of Figure 4.6 and extended by the rewritings connected with union POPs. The result is shown in Figure 4.18(b).

Although this is not the case for our two example queries, it is possible that the

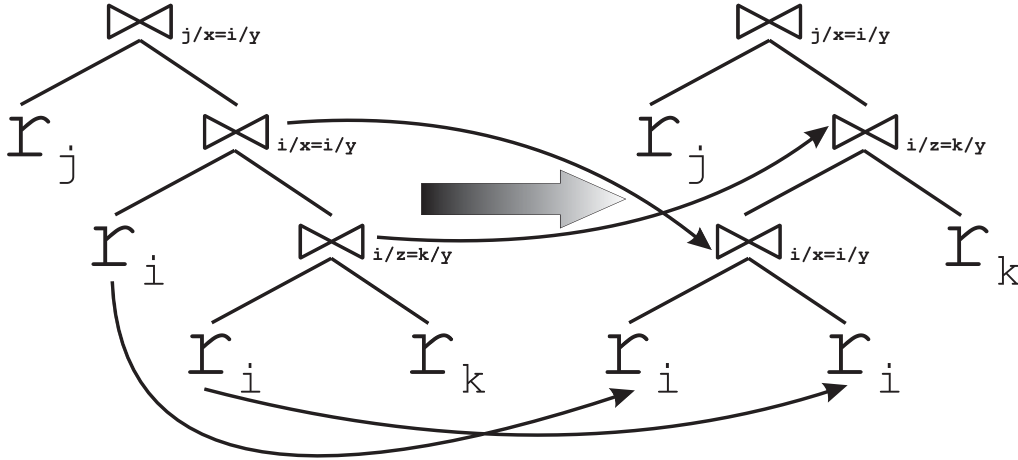


Figure 4.17: Combining Joins in a Query

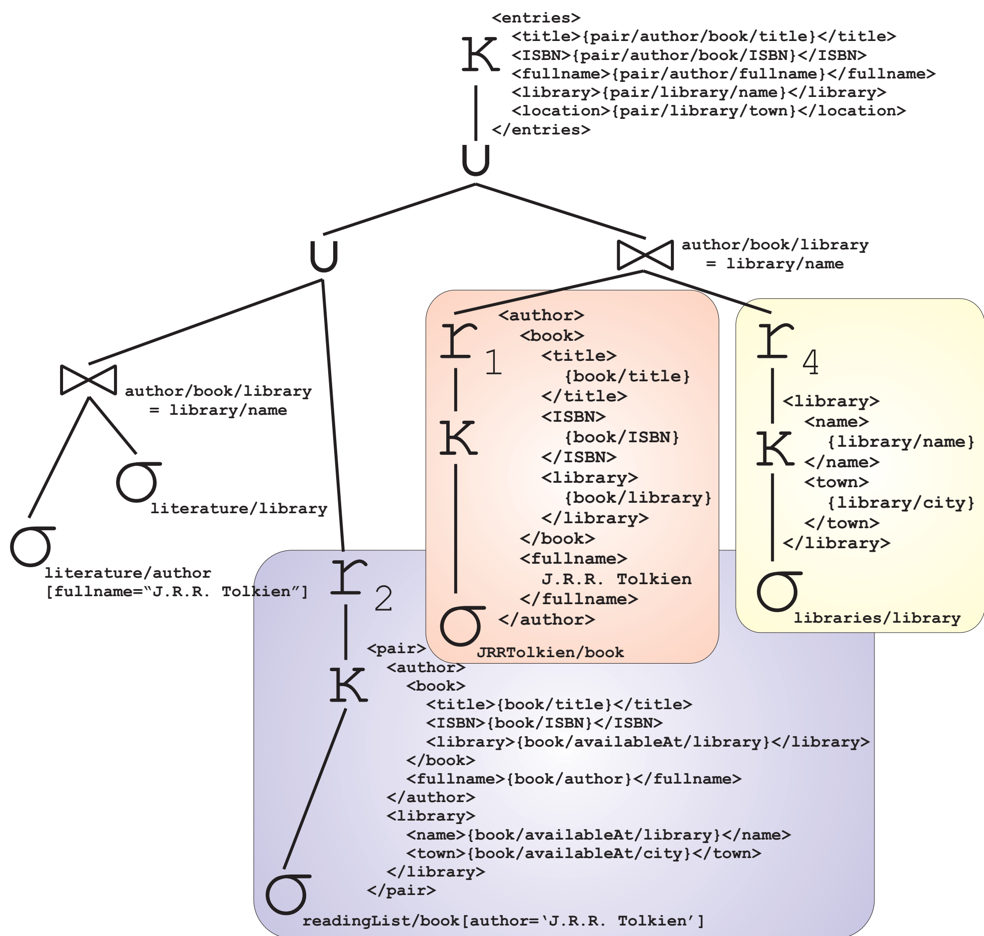
rewriting result contains two or more remote queries that are sent to the same neighbor. As each remote query is treated as an individual query, we could send all these remote queries with separate messages. However, in order to reduce network traffic, we combine all such remote queries into one message and send it to the neighbor.

Still, there is another important optimization issue. It is possible that the same remote query (originating from the same query snippet) is contained multiple times in the rewriting result. This is possible since the query snippet corresponding to a particular view subgoal of a bucket may be involved in multiple rewritings. In that case we have to identify such remote queries, have them computed by the neighbor only once, and insert the result multiple times into the query plan in order to compute the final answer to the query. To identify such remote queries we simply use the parameters of the remote query POP that we assigned to it in step 6 (Section 4.2.6). We need to look up the neighbor ID that the remote query is to be sent to and the list of subgoal/bucket IDs that the remote query fulfills.

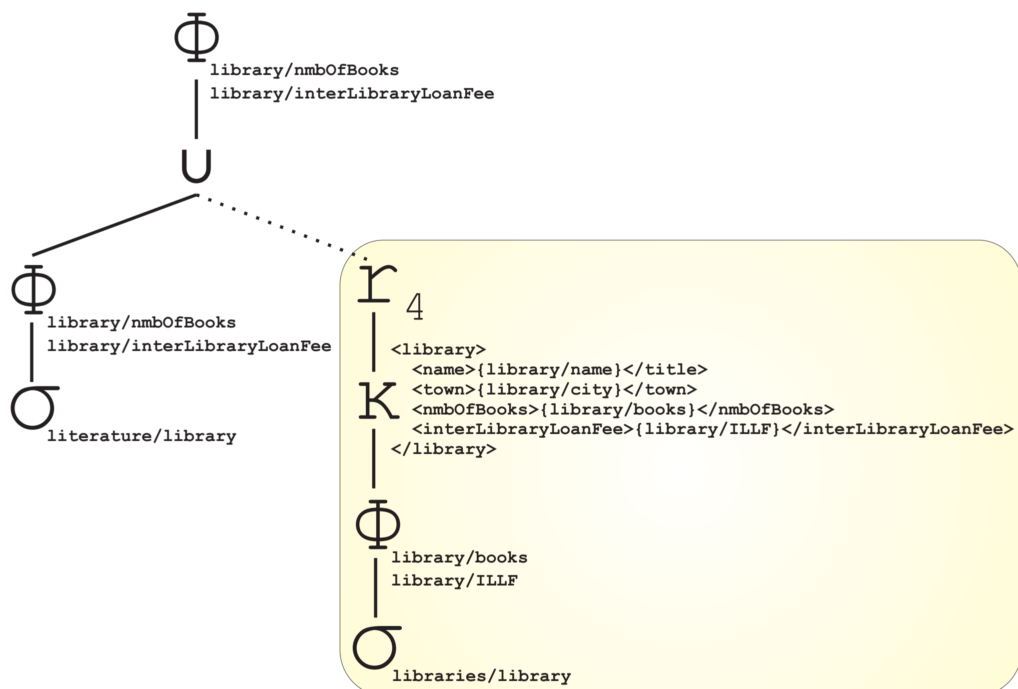
A final optimization issue, which we leave open to future work, is to check remote queries corresponding to the same neighbor for query containment and derivability. It is for example possible that the rewriting result contains a remote query corresponding to a non-altered query snippet and a remote query originating from the same query snippet but extended by an additional operator. For instance, assume the rewriting result contains the remote query shown in Figure 4.13 and a remote query, where the only difference is that the skyline operator is missing. The result of the former remote query can be obtained from the result of the latter by evaluating the additional skyline POP.

4.3 Cycle Detection

PDMSs just like P2P systems may contain cycles. Without taking any measures to avoid or break such situations, queries could in principle be sent around cycles for eternity (Figure 4.19(a)). A simple solution to this problem is to attach a time-to-live value to a query, which limits the number of times the query can be rewritten and forwarded



(a) Join Query



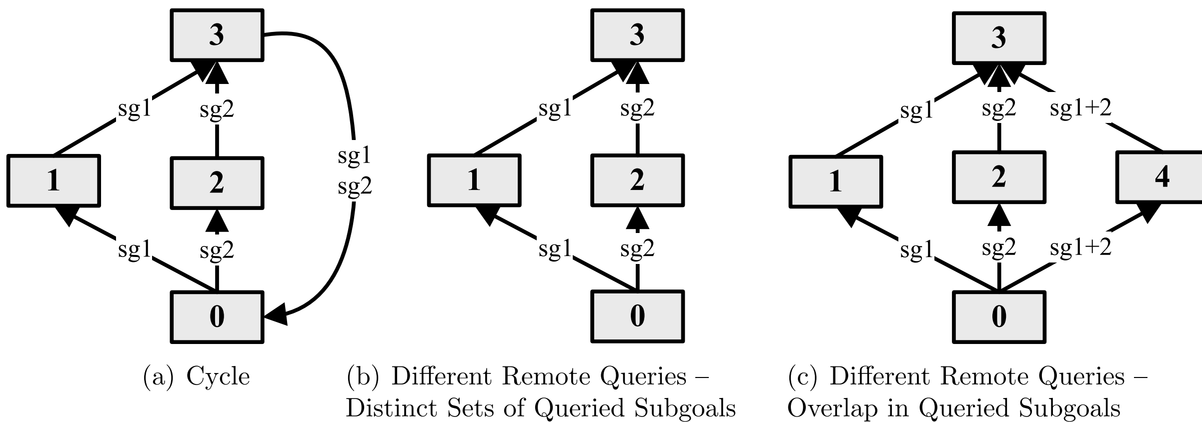
(b) Skyline Query

Figure 4.18: Rewritten Query Plans as Results of the Rewriting Process

to neighboring peers, i.e., the number of hops a query might travel. However, this should be considered only a backup solution if all other measures to detect cycles fail. Actively detecting cycles not only reduces network load but also reduces the number of duplicates contained in the answer to a query. In this section, we adopt the assumption of local completeness that we have already used to prune subgoal combinations in step 5 (Section 4.2.5).

A simple solution to detect cycles is to use a unique identifier assigned to the query message. Whenever the query is rewritten, the messages containing the remote queries are assigned the same query ID as the original query. Detecting cycles comes down to detecting whether two queries received from different neighbors have the same query ID. If yes, the peer that has received the two queries has detected a cycle. The peer then simply processes only the message received first and rejects all others with the same ID. Rejecting a message means that the query is not processed or rewritten at the peer and no answers are sent back to the sender of the query. However, the peer might still acknowledge receipt of the query and send an answer message if the query processing strategy requires to do so (e.g., Query Shipping, Section 2.5.1).

However, in PDMSs this alone is no practical solution, as the example illustrated in Figure 4.19(b) reveals. When a peer (P_0) rewrites the original query, the messages forwarded to its neighbors (P_1 and P_2) do not necessarily have to contain all subgoals of the original query. Rather, it is likely that they concern different subgoals (P_1: subgoal 1, P_2: subgoal 2). As both messages that P_1 and P_2 send to P_3 have the same query ID, P_3 only processes the first one and rejects the other one. As P_3 provides data for both subgoals, it is simply wrong to reject the second message if it concerns a different subgoal than the first one. Consequently, P_3 cannot simply reject one of the messages but instead has to answer both.



(a) Cycle (b) Different Remote Queries – Distinct Sets of Queried Subgoals (c) Different Remote Queries – Overlap in Queried Subgoals

Figure 4.19: Cycle Detection

This is a little different for the example network illustrated in Figure 4.19(c). In this network we have P_4, which also receives a message from P_0 querying both subgoals of the query. P_4 rewrites the query and sends a message concerning both subgoals to P_3. If P_3 answers all three queries, there will be a high number of duplicates in the result at P_0 because the data originating from P_3 is sent multiple times to P_0. Furthermore, local processing time at the peers and network load is unnecessarily high. To reduce

this overhead, P_3 would only have to answer the message received from P_4 and reject the other two since all data provided by P_3 with respect to both subgoals reaches P_0 via P_4. Because of using LAV-style mappings, P_3's data cannot be combined at another peer with a join that was not contained in the original query. Thus, by assuming local completeness at P_3, i.e., other peers do not provide join partners for the subgoals of P_3 that it does not store locally, the query result at P_0 would be correct.

If we cannot assume local completeness, the alternative is not to include the join in a rewriting. Instead, we would have to process the join between all subgoal combinations and all peers (as discussed in step 5, Section 4.2.5). Consequently, we would query the data of all subgoals in separate (not in a joined format) – the cycle detection algorithm would still be applicable as by using LAV-style mappings data cannot be combined with "new" joins on the path between data origin and query initiator. So, if the mappings are correct, it does not matter on which path the data is propagated to the initiator.

The problem that we still need to discuss is that in the absence of global knowledge detecting cycles is difficult. Therefore, the order in which P_3 receives the messages plays a decisive role. If P_4's message arrives first, then it is safe to reject the messages received from P_1 and P_2. In case one of the messages from P_1 or P_2 is received first, an optimal solution is much more difficult because in absence of global knowledge P_3 cannot know if it will also receive a message from P_4. Thus, it is, for instance, possible that P_3 has already processed the message from P_2. Note that processing the query at P_3 might involve some of its neighbors – not illustrated in Figure 4.19. When P_3 afterwards receives the message from P_4, it is likely that the same neighbors are queried again with respect to the same data. The best solution to the problem is to reuse the answers that have already been processed. This is possible by comparing the query of P_4 to the cached query of P_2 and identifying common remote queries. The answers to these remote queries can directly be used to answer the message received from P_4.

In case P_3 has already sent an answer to P_2, then P_0 would still have to be advised to reject all answers received from P_2 and use only those received from P_4. Although this reduces duplicates in the result set, it causes overhead. But if P_3 has not yet sent an answer to P_2 when it receives the query from P_4, then the answers (originally processed for P_2) might be redirected once they are received from the queried neighbors and used to answer the query received from P_4. P_3 would declare the query from P_2 rejected and send only one answer message to P_4.

However, so far we do not yet consider such sophisticated optimization techniques in our current implementation. Nevertheless, it is optimal when the message from P_4 arrives at P_3 first. In this case the queries received from P_1 and P_2 are rejected. If the messages arrive at P_3 in any other order, the result may contain duplicates.

We still need to discuss how P_3 might detect cycles and how it might learn that queries received from different neighbors belong to the same original query but concern different subgoals. In order to reconcile cycle detection using message IDs with the problems that we have identified, we enhanced the concept of message IDs. Whenever the query is rewritten into several remote queries, the message ID of the original query (O_{ID}) is extended by the subgoals that the remote query, which is assigned the new message ID, fulfills. A remote query is only assigned the original non-extended message ID if it fulfills all subgoals of the original query. The remote query POPs contain all information that is necessary to extend the message ID: the subgoal IDs, the total number of subgoals of

the original query, and the ID of the peer that receives the remote query. With respect to this information, the message ID (M_{ID}) of remote queries is created as follows:

$$M_{ID} = O_{ID} + \text{"\#"} + PeerID + \text{"_"} + subgoal_{ID}$$

Applied to the example of Figure 4.19(c) and an original message ID of $0-1-2$, P_0 would send messages with the following IDs:

$0-1-2 \# 0 _ 1$ (to P_1),
$0-1-2 \# 0 _ 2$ (to P_2), and
$0-1-2$ (to P_4).

Consequently, P_3 receives messages from its neighbors with different message IDs so that the problem that we had with the original understanding of message IDs is solved. A cycle can still be detected by comparing the message IDs of the received messages. Since the original message ID is always preserved as a prefix, cycles can be detected by checking if the cache stores (i) a message with the same ID or (ii) a message with the same prefix. Assume P_3 in Figure 4.19(c) receives two messages originating from P_0 with the following IDs: $0-1-2 \# 0 _ 1$ and $0-1-2 \# 0 _ 2$. P_3 now checks if there are any messages in its cache with the same IDs (there are no such messages). But there is one message with ID $0-1-2$ (received from P_4). Since this is a prefix to both received messages, P_3 has detected a cycle and rejects the received queries.

4.4 Evaluation

In the previous sections we presented a query rewriting strategy for use in PDMSs. In this section we discuss our evaluation results [94, 165] that we obtained by integrating the presented techniques into SmurfPDMS (Chapter 7). As this evaluation aims at query rewriting, we did not apply any pre-selection of "relevant neighbors" based on routing indexes. Thus, in our evaluation all mapping definitions to neighboring peers that a peer knows are considered in the rewriting process.

4.4.1 Rewriting vs. Schema Indexing

In our first tests we examined the application of our techniques in comparison to a network that does not use query rewriting. For this purpose, we used the network of 7 Peers of Figure 4.20 – providing altogether 42 data records in the schema of P_0 but distributed among peers in the network.

It is obvious that in such a scenario an approach that does not consider any rewriting at all would never be able to find all result records. The reasons are heterogeneity and the inability to exploit schema correspondences. Thus, the query would have to be forwarded in its original form, which means formulated on the initiator's schema. As schema elements that refer to the same real world entity are named differently in the network of Figure 4.20, we chose to run the non-rewriting strategy on a slightly adapted network, which is depicted in Figure 4.21. In this network, all elements referring to the same real-world entity are named the same among all peers. Furthermore, we decided to give the non-rewriting strategy a little help in doing some schema-level pruning: each

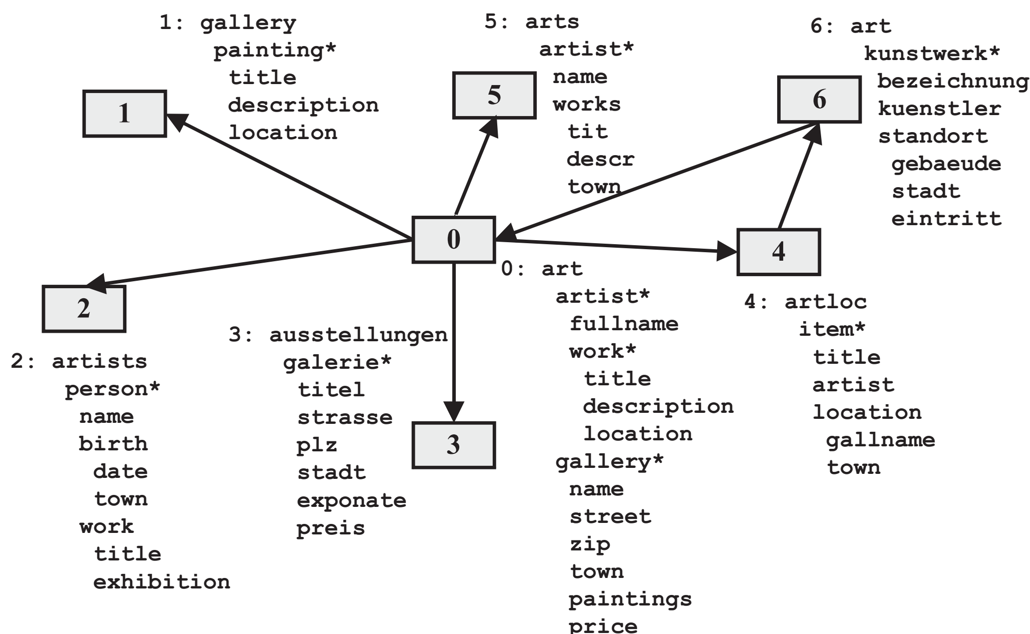


Figure 4.20: Example Network with Mappings

peer stores a routing index that indexes schema elements – indexing element names in the sense of keywords. This means a peer has information about the existence of schema elements that can be accessed by forwarding a query to a specific neighbor.

Thus, in summary we have two scenarios: (1) the network of Figure 4.20, where we apply the rewriting techniques presented in this chapter and (2) the non-rewriting approach on the adjusted network of Figure 4.21 with schema indexes. In total, we issued 6 queries on P_0, which are listed in Table 4.1. To give both test scenarios the same chances, the queries only refer to schema elements shared by both networks.

QueryID	**Query Type**	**#Levels in POP tree**
0	Projection	1
1	Projection and Transformation	2
2	Projection and Selection	2
3	Join with Transformation	3
4	Top-N with Transformation	3
5	Skyline with Transformation	3

Table 4.1: Evaluation Query Mix

The most important cost factors in distributed environments are network load (data volume) and the number of messages that are necessary to answer a query. Thus, we discuss our evaluation results using these two measures and neglect local execution costs. Let us first consider a basic non-incremental query processing strategy (Query Shipping, QS, Section 2.5.1). With respect to network load in terms of transferred data volume,

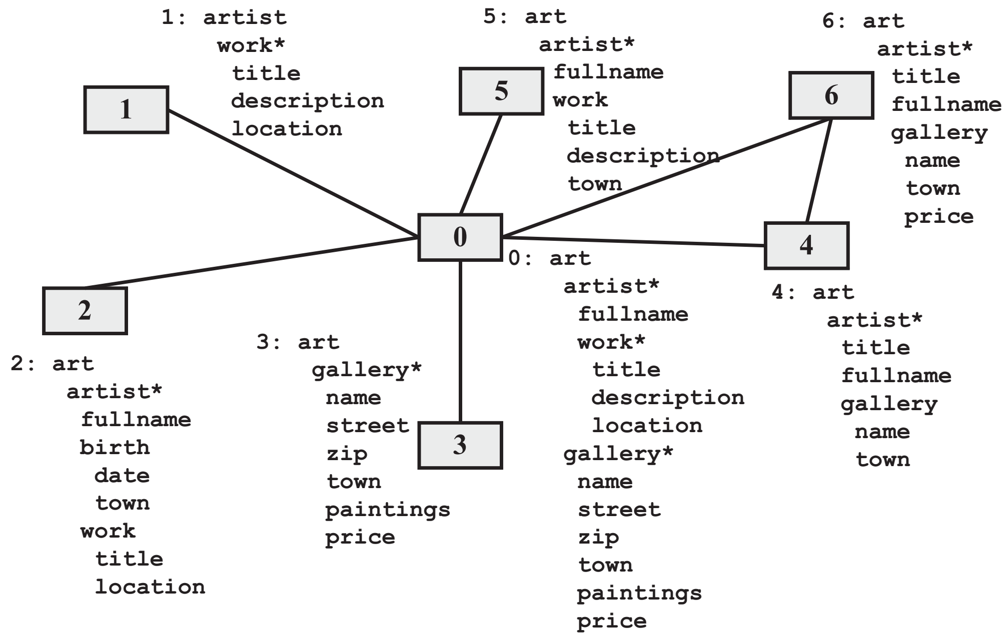


Figure 4.21: Example Network without Mappings

the rewriting strategy should produce less load since in contrast to the non-rewriting strategy only remote queries are sent to neighboring peers instead of the whole query. With respect to the number of messages, we expect more or less the same results since both scenarios use schema information to identify relevant neighbors.

The results of our tests are listed in Table 4.2. It shows the total volume and the total number of messages that were necessary to answer all 6 queries of the query load. As we have expected, data volume is reduced by our rewriting techniques. Furthermore, also the number of messages is reduced. This is due to the fact that the rewriting strategy is able to prune a few more neighbors because only those neighbors are queried that provide all elements that are necessary to answer a query or a subgoal, respectively.

	Scenario 1 – Rewriting	**Scenario 2 – No Rewriting**	**Difference**
Total Number of Messages	50	68	$-26,5\%$
Data Volume in kByte	133.053	241.568	$-44,9\%$

Table 4.2: Results for Query Shipping

We did the same tests applying a basic incremental query processing strategy (Incremental Message Shipping, IMS, Section 2.5.3). In contrast to QS, peers no longer have an obligation to send an answer message for each received query message. Remember the application of QS requires that each peer sends an answer message for each query

message it receives such that the sender can stop waiting for answers. Thus, the difference between the two test scenarios should be less obvious than in the tests for QS. Still, in scenario 2 more queries should be forwarded to neighboring peers due to the same reasons as stated above. However, without the obligation to answer each received query the impact of asking some additional peers (i.e., peers that do not contribute to the result) should be smaller than in the QS tests. The data volume should show the same tendencies as for QS.

The test results shown in Table 4.3 support our anticipation. The difference between scenarios 1 and 2 is smaller than it is in Table 4.2. Furthermore, in comparison to QS, IMS needs fewer messages to answer the query since there is no obligation to send answer messages.

	Scenario 1 – Rewriting	**Scenario 2 – No Rewriting**	**Difference**
Total Number of Messages	49	54	$-9,2\%$
Data Volume in kByte	135.581	205.407	$-34,0\%$

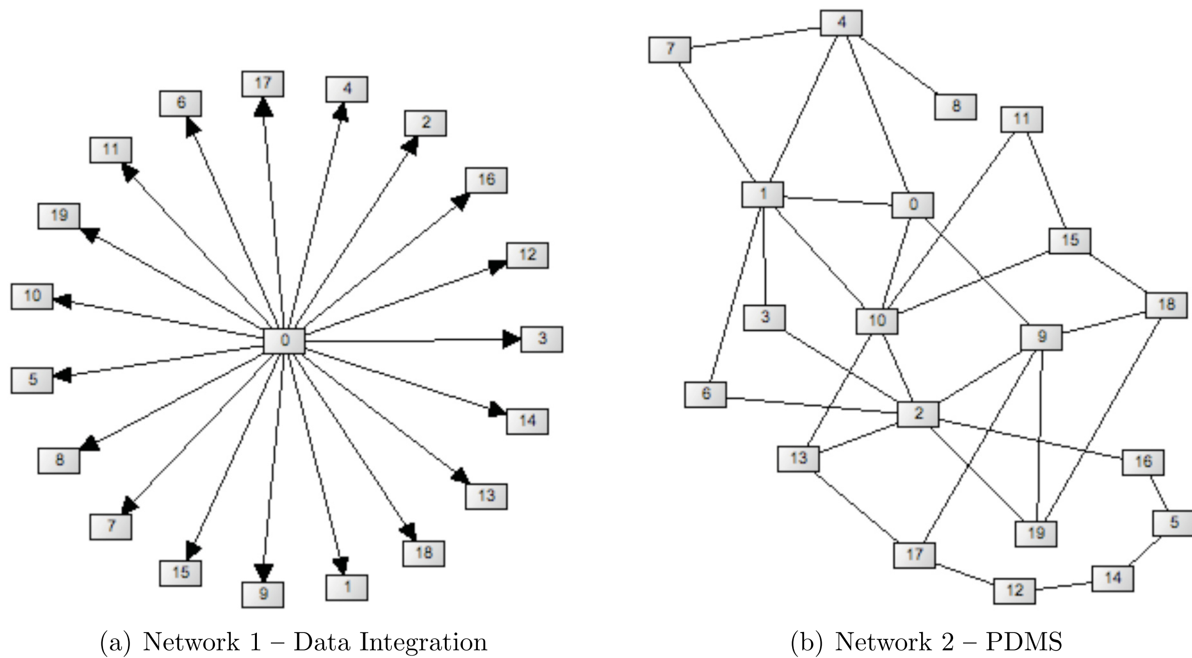
Table 4.3: Results for Incremental Message Shipping

Our first results show that the rewriting method is preferable over a non-rewriting strategy since the knowledge contained in the mappings allows for a more effective schema-level pruning. Another positive effect of using mappings is that a peer is able to query schemas differing from its own. In comparison, the schema index approach we used in scenario 2 can only be used in rare situations. One of them is a scenario where the local data of the peers participating in the network originates from vertical and horizontal partitioning of a common source database. In that case all peers have the same naming and structuring of elements so that a query formulated at one peer can be interpreted correctly by other peers in the network. However, such a setup is not the usual case.

4.4.2 The Influence of the Choice of Neighbors

Query processing in PDMSs strongly depends on the choice of neighbors, i.e., it depends on which neighbors a peer has established mappings to. The greater the distance to the sources in the network that hold relevant data for a particular query, the higher is the chance of missing such data. The reason is that mappings are often incomplete. Hence, distant data that matches the query can only be considered for processing a particular query if all elements referred to by the query are contained in all rewritings of peers on the path between initiator and data provider. In order to show that our techniques still work in a scenario where a query has to be rewritten more often than just once, we created two networks each consisting of 20 peers (Figure 4.22). The data of these peers (300 data records) can be divided into four topics: galleries, artists, art objects (paintings or sculptures), and styles.

In scenario 1 (Figure 4.22(a)) the network structure corresponds to a star with P_0 having connections to all other peers. Thus, scenario 1 represents a standard data inte-



(a) Network 1 – Data Integration

(b) Network 2 – PDMS

Figure 4.22: Example Network with 20 Peers

gration system where P_0 serves as mediator and provides a common global schema using LAV-style mappings. In scenario 2 (Figure 4.22(b)) we arranged the peers in clusters determined by a similarity measure, which is defined as the number of possible schema element correspondences between the schemas of any two peers. For each link we defined bidirectional mappings, i.e., if P_0 has a mapping to P_1, then P_1 also has a mapping to P_0. While this increases data reachability, it also increases the number of cycles that have to be dealt with. Because of the clusters, there are many links between peers providing similar data, i.e., data of the same topic. Since some of the peers store data of more than just one topic, clusters overlap. In order to answer queries that involve joins between several topics correctly, we established additional links that interconnect clusters.

In both scenarios we issued 13 queries (two involving top-N and skyline operators) with at most 4 subgoals at P_0 applying Query Shipping (QS, Section 2.5.1). Although queries need to be rewritten more than just once in scenario 2, results (disregarding duplicates) in both scenarios should be the same due to the favorable choice of neighbors. In addition, we expect duplicates in the result set of scenario 2 as it is possible that a peer receives remote queries from different peers that belong to the same query but arrive in an unfavorable order, as discussed in Section 4.3.

Table 4.4 shows our results with respect to the two scenarios, summarizing the results for all test runs and queries. In both scenarios all answers to the query have been retrieved. Since some data records are stored at multiple peers, both scenarios retrieve a small number of duplicates that the rewriting strategy cannot detect. As anticipated, in the second scenario we retrieve more duplicates due to the reasons indicated above.

As our tests have shown, when optimal neighbors are chosen with respect to (i) the similarity of schemas and (ii) the combinability of neighbor data (subgoals) for join

Scenario	Result Size	Result Size after Removing Duplicates
Data Integration	304	294
PDMS	353	294

Table 4.4: Comparison of Query Processing in Data Integration Systems and Distributed Query Processing in PDMSs

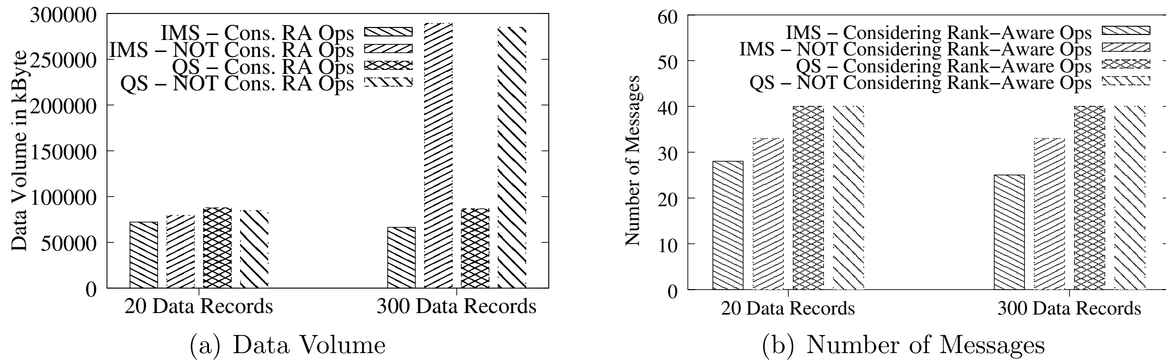
operations, all the data in the system can be accessed. Performance can still be improved by elaborating cycle detection and influencing the network topology.

4.4.3 Benefits of Considering Rank-Aware Query Operators for Rewriting

In our tests we also examined the benefits of considering rank-aware query operators in the rewriting process. Since we have designed our algorithms for the purpose of reducing costs by considering such operators, we expect the costs to be lower in comparison to a rewriting strategy that does not consider such options. For comparison, we used the implementation of our rewriting strategy in SmurfPDMS and bypassed all optimizations that consider rank-aware operators. For our tests we used the PDMS network of 20 peers illustrated in Figure 4.22(b). We chose a 3-level top-N query (*topn* POP, *construct* POP, and *select* POP) with $N = 5$ and issued it twice for each of the two modes at the same peer (P_0) – once using QS and once using IMS. In our tests, we also varied the number of records whose structure fits the query: 20 and 300 data records distributed among all peers.

The results of our tests are shown in Figure 4.23. As we have expected, in general message volume (Figure 4.23(a)) is reduced when considering rank-aware operators for rewriting. The savings for a low amount of relevant data is smaller than for high amounts. The reason simply is that if the result size is larger, more records can be pruned at an early stage at the peer providing the data so that they are not sent to the initiator just to be pruned there. Consequently, data volume is reduced. In general, IMS performs slightly better than QS in terms of data volume because of the lower number of messages (Figure 4.23(b)).

The number of messages is primarily determined by the number of peers whose schemas "match" the query. However, applying IMS a peer is not forced to send answer messages to the peer it has received the query from. Thus, if a peer decides to reject a message or if it has no local answers to the query, it does not send a message so that the number of messages in comparison to QS is lower. This effect is amplified by the application of rank-aware query operators. The reason causing this effect is that a top-N operator contained in a remote query, which is sent to a neighboring peer, contains the partial result of the sender or rather the worst record that is part of its local result (Chapter 6). The receiver then uses the additional information to decide whether it might contribute to the final result. If all its local data records are ranked worse than those received along with the query, the peer does not send an answer message, which it



(a) Data Volume

(b) Number of Messages

Figure 4.23: Benefits of Considering Rank-Aware Query Operators for Rewriting

would have done if it had not known about the local result of the query's sender.

Finally, let us mention that, although we do not go into detail on the results, the same findings not only hold for top-N operators but for skyline operators as well.

4.4.4 Costs for Rewriting

So far we have only considered network load and neglected the computational load caused by query rewriting. In this section we examine all nine steps of our rewriting algorithm (Section 4.2) with respect to their execution costs.

In the first step (receiving a query plan), we only need to traverse the query POP tree once in order to check conformance to the requirements. As we need to visit each POP of the tree at most once, this results in costs $O(p)$ with p being the number of POPs contained in the query POP tree. Note that we need to do this check only once at the initiator of the query because all other peers receive queries generated by our algorithm and those queries always fulfill the requirements. Thus, there is no need to check them again. As a query usually consists of only a few POPs ($p < 10$), the costs for the first step can be neglected resulting in $O(1)$.

In the second step (preprocessing), we have to consider union POPs and rank-aware POPs. In step one we have already visited all POPs and still remember all union and rank-aware POPs contained in the query. Thus, locating them now is done in $O(1)$. Queries containing union POPs are split up into several subqueries such that none of them contains union POPs. This can be done in $O(1)$. Furthermore, preparing the rewriting result for rank-aware operators, i.e., cloning the rank-aware operator and inserting it on top of the rewriting result, is also done in $O(1)$. As the number of union and rank-aware POPs in a query is rather low, the costs for the second step can be neglected, too.

In the third step (creating buckets), we need to create buckets for each subgoal contained in the query. Consequently, the effort is $O(n)$ – with n denoting the number of query subgoals.

In the fourth step (sorting view subgoals into buckets), we need to compare each view subgoal to each query subgoal. The costs for a total of m view subgoals are:

$$O(n * m)$$

In the fifth step (creating combinations of buckets), we at first create the Cartesian product between all view subgoals that have been sorted into the buckets: costs are $O(c_{max})$ with c_{max} denoting the maximum number of combinations. We obtain the maximum number of combinations c_{max} if all m view subgoals have been sorted into all n buckets:

$$c_{max} = m^n$$

A query as well as a view contains multiple subgoals only if it refers to different base structures of the data. Thus, it is only possible that all view subgoals fulfill all query subgoals if $n = 1$ and $m = |sg(\mathcal{V})|$ – with $sg(\mathcal{V})$ denoting the set of view subgoals of all views that are considered for rewriting. In any other case, the number of combinations is maximal if the view subgoals are uniformly distributed among all buckets:

$$c_{max} = \prod_{i=1,\ldots,n} m_i, \;\; m_i = \frac{m}{n}$$

and thus:

$$c_{max} = \left\lceil \frac{m}{n} \right\rceil^n$$

Afterwards, the number of combinations is reduced so that all combinations that do not fulfill Equation 4.1 (removal of redundant subgoal combinations) are removed and not considered any further. Thus, the remaining combinations consist of views whose sets of fulfilled view subgoals do not overlap. In order to check this, we need to access each subgoal of every combination. The costs are:

$$O(c_{max} * n)$$

For all remaining combinations we need to check whether the predicates are contradictory ($O(1)$). Checking this for every pair of view subgoals in a combination, we have to do

$$c_{max} * \binom{n}{2} = c_{max} * \frac{(n-1)*n}{2} = c_{max} * \frac{n^2 - n}{2}$$

comparisons. Note that we do not make any assumption on how many combinations get pruned but use the worst case scenario for our estimations. In most cases, the costs are lower as at least some of the combinations get pruned either because of contradictory predicates or Equation 4.1. In total, the costs of step five amount to:

$$\begin{aligned} O\left(\left\lceil \frac{m}{n} \right\rceil^n + \left\lceil \frac{m}{n} \right\rceil^n * n + \left\lceil \frac{m}{n} \right\rceil^n * \frac{n^2 - n}{2}\right) &= O\left(\left\lceil \frac{m}{n} \right\rceil^n + \left\lceil \frac{m}{n} \right\rceil^n * \frac{n^2 + n}{2}\right) \\ &= O\left(\left\lceil \frac{m}{n} \right\rceil^n * \frac{n^2 + n}{2}\right) \\ &= O\left(\frac{(m+n)^n}{n^{n-2}}\right) \end{aligned}$$

In the sixth step (creating query snippets), one query snippet is created for each view subgoal that has been sorted into a bucket. This only depends on the total number of view subgoals m. However, it is unlikely that all view subgoals of all views are relevant to a query. Nevertheless, the costs for this step are in $O(m)$.

In the seventh step (creating initial rewritings), a rewriting for each combination that has not been pruned is created. Consequently, the costs are in $O(c_{max})$.

For each rewriting we try to optimize it (in the eighth step) and push as many operators as possible down underneath the remote query POP. The costs for optimizing a specific rewriting are determined by the number of operators that could be pushed down, which again is determined by the number of POPs p the original query consists of. Thus, the corresponding costs for optimizing a specific rewriting are in $O(p)$. Consequently, the costs for optimizing all rewritings are in: $O(c_{max} * p)$. As p is a constant factor, which is only in very rare cases greater than 10, we can neglect p and state that the costs for optimizing all rewritings are in $O(c_{max})$.

In the ninth and last step (assembling the rewriting result), the final output query plan is assembled. For this purpose, we need to access each rewriting once. Consequently, costs are in $O(c_{max})$.

In total, the worst case running time for rewriting a query are in the worst case in:

$$\begin{aligned} Costs &= O(1) + \\ &\quad O(1) + \\ &\quad O(n) + \\ &\quad O(n * m) + \\ &\quad O\left(\frac{(m+n)^n}{n^{n-2}}\right) + \\ &\quad O(m) + \\ &\quad O\left(\left(\frac{m}{n}+1\right)^n\right) + \\ &\quad O\left(\left(\frac{m}{n}+1\right)^n\right) + \\ &\quad O\left(\left(\frac{m}{n}+1\right)^n\right) + \\ &= O\left(\frac{(m+n)^n}{n^{n-2}}\right) \end{aligned} \tag{4.2}$$

These costs seem to be rather high. However, in general there is only a small number of query subgoals because a query is only in rare cases defined on more than just a few base structures. Thus, the most important factor that determines costs is the number of view subgoals that need to be considered for rewriting. As we have already discussed in the beginning of this chapter, the application of routing indexes might help to prune several neighbors from consideration. Hence, if we consider routing indexes before rewriting the query, we can reduce the number of view subgoals that need to be considered by the rewriting component. Consequently, we decided to have peers consider routing indexes before rewriting the query.

4.5 Conclusion

In this chapter we presented a query rewriting strategy for use in PDMSs. It addresses the problem of rewriting queries containing rank-aware operators formulated as POP

trees in conjunction with LAV-style mappings and XML data. Our evaluation has shown that by integrating rank-aware operators into rewritings, network load can considerably be reduced. The higher the number of records that are relevant to a query, the more reduction in execution costs can be achieved by applying our techniques.

Although our techniques perfectly solve the problem we designed them for, there are several interesting aspects that might be considered in future work. One of them is the integration of GLAV-style mappings and complex mappings in general that support not only 1-1 correspondences but also 1-n, n-1, and perhaps even n-m correspondences. Furthermore, the use of a query cache in conjunction with testing query containment and issuing compensation queries is another option that has the capability to reduce execution costs even more. Moreover, so far we consider only mappings (or view subgoals) for rewriting that provide matches for all attributes defined in a query. In this context, it might be worthwhile to develop a "best effort" rewriting strategy which also considers view subgoals that do not provide matches for all queried attributes. This might be an interesting strategy to counteract the problems we have to deal with in PDMSs, e.g., incomplete mappings. Our evaluation has shown that the network topology affects execution costs. Thus, the "right" choice of neighbors plays a decisive role. Therefore, the application of peer clustering strategies and mapping composition [61, 210], maybe even (semi-)automatic schema matching [167], seems to be a worthwhile effort.

Chapter 5

Query Routing

As already pointed out in the introduction, efficient query routing is a crucial part of any query processing strategy in P2P systems. Routing in this context does not mean finding a route to a specific peer (peer-based) but routing queries to peers that hold relevant data (data-based). The most primitive data-based routing approach in unstructured P2P systems is flooding, i.e., forwarding the query to all the peers in the system and have each peer send its local result directly to the initiator. Once the initiator has received and aggregated all answers, the result can be output to the user. It is obvious that this cannot be efficient in the general case since the initiator has to bear a very high workload. Most important is the fact that this strategy is not applicable to PDMSs, where peers can only communicate with peers they have already established mappings to. Still, this technique can be improved and made applicable to PDMSs by routing the answers on the same paths that the query has been forwarded on. By pre-aggregating the answers on their way to the initiator, load at the initiator is reduced.

Nevertheless, there is another improvement that even avoids expensive flooding of the network: routing indexes can be used to route queries to only those peers that provide relevant data. For instance, if the routing indexes indicate that a specific peer does not provide any relevant data to a given query, then it is reasonable to prune the peer from consideration (data-level pruning). This reduces network load as well as the number of involved peers and still provides the user with the correct answer. Each peer has to decide independently from all the others and with the help of its routing indexes (and information received along with the query) whether the data accessible via each of its neighbors is relevant to the query or not. By identifying attributes and constraints (formulated in the query) that describe relevant data, a peer can use its routing indexes to identify relevant and irrelevant neighbors.

Another option to reduce query execution costs is to apply rank-aware query operators (top-N and skyline) or approximation. However, their application can only be considered by any query processing strategy if routing indexes are supported. As introduced in Section 2.3, in their original sense routing indexes were used to index keywords. However, in subsequent works this concept has been extended and applied to numerical data. This is how we make use of routing indexes as well. In this dissertation we follow the understanding of routing indexes as given in Definition 2.3.1 and identify a subclass of such routing indexes – *Distributed Data Summaries* (*DDSs*). We limit our considerations to compound routing indexes (CRIs) with respect to the classification of Figure 2.7. We

will formally define DDSs in Section 5.1, propose a novel base structure for routing indexes [90, 213] in Section 5.2, and focus on its maintenance and construction [118, 213] in Section 5.3. Figure 5.1 illustrates the importance of routing indexes, or distributed data summaries respectively, in the general steps of query processing in PDMSs that we have identified in Section 1.2.

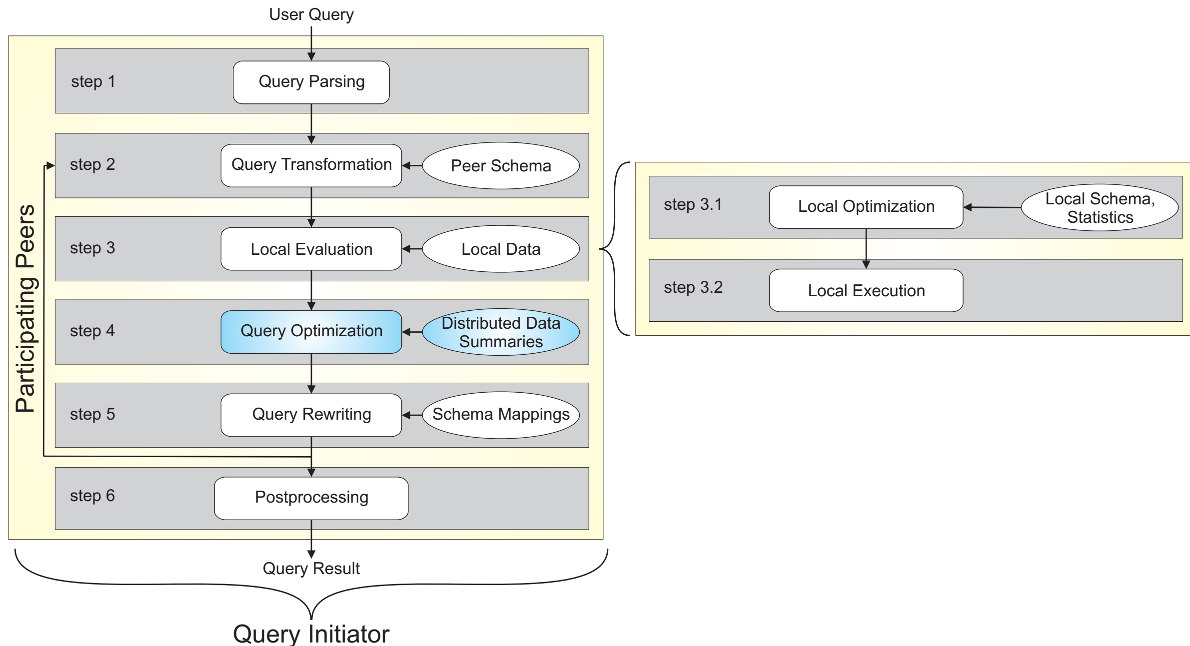


Figure 5.1: Query Processing in PDMSs – Distributed Data Summaries

5.1 Distributed Data Summaries

Each routing index at a peer represents information about the data provided by neighboring peers. This information is not limited to the data the neighbors store locally but also includes the data that can be accessed via their neighbors, e.g., data located several hops away. Assume a routing index keeps the information for each neighbor in a separate instance of a base structure so that it is possible to identify unambiguously what data is accessible via each neighbor. Without loss of generality, assume that a routing index complies with the structure illustrated in Figure 5.2.

The choice of the base structure is crucial to performance and usability of any routing index. Appropriate base structures for use in our application scenario have to fulfill several requirements. First of all, such a structure has to summarize the data in order to minimize/limit memory consumption. This is important because there is only a limited amount of memory space available. Furthermore, non-summarizing indexes such as B-trees [13, 14] and R-trees [83, 143] would require far too much disk space and would be hard to maintain efficiently when peers update their data. However, at the same time the structure should minimize the approximation error, which is the logical consequence of data summarization. Another requirement is that the base structure has to support efficient lookups as this is what we will need it for. Furthermore, it should be possible to

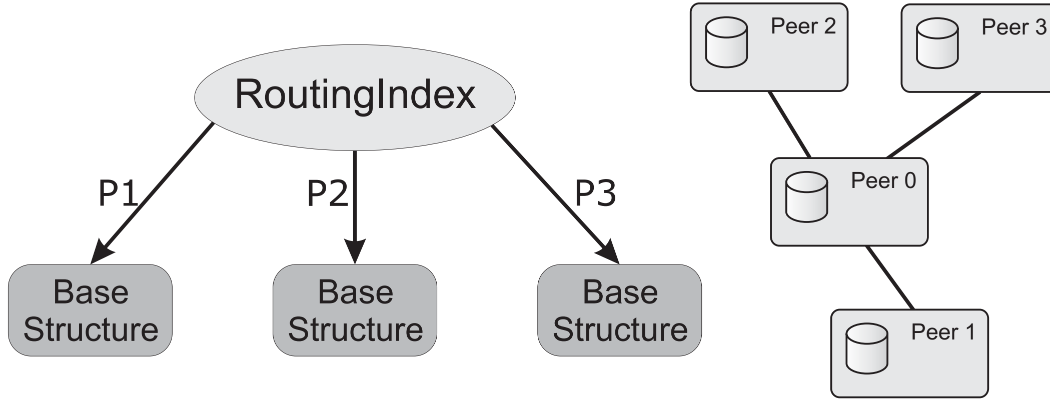


Figure 5.2: Structure of a Routing Index at P_0

update the structure because in dynamic systems we have to assume that peers establish new links or update their local data. It should be possible to perform incremental updates without having to create a new instance of the structure whenever an update occurs.

As we have already pointed out in Section 2.3, it is very important to use a base structure that captures attribute correlations. The reason is that in contrast to routing indexes that only describe attributes in isolation, the number of false positive routing decisions can be reduced by far – false positive in this context means that a peer is classified as relevant to the query when actually it does not provide any relevant data. As we aim at processing queries defined on multiple attributes (especially skyline queries), this requirement is very important. Furthermore, routing indexes should never provide information that might lead to false negative routing decisions, i.e., classifying a peer as irrelevant although it provides relevant data. Finally, routing indexes should be all-purpose. This means they should be useful to all queries that a peer might issue. Consider for example skyline queries. If we precomputed a local skyline at each peer and indexed only that data, we could very efficiently process and route skyline queries [196]. However, we could not use the same structure to identify relevant peers for any other query type. The alternative of holding several routing indexes for several query types is inefficient. We argue that it is more worthwhile to use the additional disk space to create just one but more detailed general-purpose routing index, e.g., by allowing a histogram-based routing index to use more buckets. Thus, the last requirement, which should be fulfilled by a routing index, is that it should be beneficial for all queries and not restricted to any specialized type.

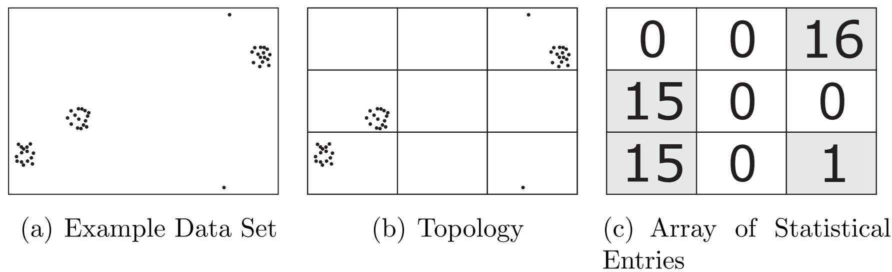
Routing indexes whose base structures fulfill these requirements are referred to as Distributed Data Summaries (DDSs) according to Definition 5.1.1. In the following, we use the terms routing index and DDS as synonyms.

Definition 5.1.1 (Distributed Data Summaries (DDSs))**.** *A distributed data summary is a data summarizing structure that captures information about the data accessible via all neighboring peers. Each peer maintains exactly one such structure. A distributed data summary holds one summarizing base structure for each neighbor such that it meets the following characteristics:*

- *summarizing data while restricting memory consumption and minimizing the approximation error at the same time,*

- *supporting efficient lookups,*
- *capturing attribute correlations to efficiently process multidimensional data and queries defined on multiple dimensions (attributes),*
- *supporting (incremental) updates, and*
- *being all-purpose so that there is no restriction to any specialized query type.*

As we have already discussed in Section 2.3, multidimensional histograms do support all these requirements. Thus, let us sketch an example for DDSs based on histograms. Figure 5.3 illustrates how an equi-width histogram can be used to represent a two-dimensional example data set. Figure 5.3(a) shows the data set that is to be indexed, i.e., the data accessible via a particular neighbor. Figure 5.3(b) shows the corresponding grid that originates from partitioning the data space into buckets. Finally, Figure 5.3(c) shows the resulting two-dimensional array of statistical entries, i.e., the number of records contained in the buckets.



(a) Example Data Set (b) Topology (c) Array of Statistical Entries

Figure 5.3: Equi-Width Histogram as Base Structure

There is another interesting class of structures that is often used to organize and accelerate access to large data sets. Most of these approaches are based on the B-tree [13, 14] or on the R-tree [83, 143] as its multidimensional extension. Although this class fulfills most of the requirements, it lacks the support of one important requirement: summarizing data. Traditional tree structures do not approximate the data, rather they keep information about each single data record. Although this is necessary to accelerate access to local data, in the context of routing indexes we need summarizing structures as keeping detailed information requires too much memory space. Thus, we developed the QTree [90, 213], which aims to combine the advantages of tree structures with those of histograms.

5.2 QTree

As a combination of histograms and R-trees the QTree inherits not only the benefits of histograms but also the beneficial characteristics of trees: indexing multidimensional data, efficient look-ups, efficiently dealing with sparse data, and supporting incremental construction and updates. In the following we first define the structure of a QTree. Then, we describe how it is constructed. Although we are focusing on positive integer values, the extension to negative values as well as to rational numbers is straightforward and not discussed in the following.

5.2.1 Definition

In principle, the QTree is similar to an R-Tree where subtrees have been replaced with statistical nodes. Thus, a QTree is a tree structure that consists of nodes, each of which being defined by a minimal bounding box. Such a minimal bounding box (MBB) describes the region in the data space that is represented by a node and the subtree underneath. Consequently, a child node's bounding box is completely enclosed by the box of its parent. As a standard configuration we assume bounding boxes to be rectangular. However, in principle other shapes could be used as well.

In order to limit memory and disk space, we replace subtrees with special nodes – called *buckets* in analogy to histogram buckets. Buckets are always leaf nodes and leaf nodes are always buckets. Only buckets contain statistical information about the data records contained in their MBBs. The smallest possible bucket consists of only one record. Consequently, its MBB has no extension. Although buckets may contain almost any kind of statistical data, we will only consider buckets capturing the number of data records contained in their MBBs. Each QTree is defined by two parameters:

- f_{max}: the maximum fanout (number of child nodes) an inner tree node may have
- b_{max}: the maximum number of buckets in a QTree – limiting the total number of buckets that a QTree might use to approximate the data and thus limiting the size and memory consumption

Definition 5.2.1 formally defines a QTree.

Definition 5.2.1 (QTree)**.** *A* QTree *is a tree structure that consists of nodes. Each node*

- *is defined by a multidimensional bounding box (MBB) describing a region in the data space. The MBB of an inner node n completely encloses all MBBs of all descendants.*
- *has at most f_{max} children but only one parent node (except of course the root node).*
- *that has no children is called* bucket *and contains statistical information, e.g., the number of records contained in its MBB.*

A QTree may use at most b_{max} buckets.

Note that the size of a QTree ($O(b_{max})$) is independent from the number of represented records, it only depends on b_{max} (and f_{max}). In the following, we describe how a QTree is constructed and which heuristics we use to prevent it from degenerating into a linear list.

5.2.2 Construction

The construction of a QTree relies on three basic algorithms that we want to discuss in the following, these are:

- inserting a data record,
- reducing the number of buckets, and
- reducing the fanout of inner nodes.

Inserting a Data Record

The QTree is constructed incrementally by inserting one data record after another. At first, we need to check whether data record p could be inserted into an already existing bucket. Thus, if a bucket is found whose MBB encloses p's coordinates, p is inserted into the bucket by incrementing its statistical entry by 1. If such a bucket does not exist, method *insertDataRecord* (Algorithm 4) is called at the root node of the QTree. By following the algorithm, the node at which the method has been called determines a child node c whose MBB encloses p's coordinates. If such a child node can be found, Algorithm 4 is recursively called at c (lines 2–3). In contrast to R-trees, we are looking only for a child node whose MBB encloses p's coordinates. If there is no such child node, we do not try to enlarge the MBB of an existing node. Instead, we create a new bucket (with no extension and representing only p) and insert it as a a new child node (lines 5–7). As we have created a new bucket, we need to update the QTree's bucket counter, which keeps track of the number of buckets the QTree currently uses (line 8). Afterwards, the node's MBB is enlarged to enclose p's coordinates (line 9). This is only necessary if the bucket has been inserted as a direct child of the root node. Finally, we have to check if the constraints defined by f_{max} and b_{max} are still met and enforce them if necessary (lines 10–17). For this purpose, a QTree maintains a priority queue, which has to be updated each time a bucket is added or removed (line 13). As the algorithms that enforce the constraints take care of updating the priority queue, it only needs to be updated explicitly in Algorithm 4 if these constraints are not violated.

Algorithm 4 *insertDataRecord(p)*

```
1: c = getMostResponsibleChild(self.childNodes, p);
2: if c ≠ null then
3:   c.insertDataRecord(p);
4: else
5:   /* insert p as a new bucket */
6:   newBucket = new QTreeBucket(p);
7:   self.addToChildren(newBucket);
8:   root.cntBuckets++; /* update statistical entry */
9:   adaptMBB();
10:  if #self.childNodes > f_max then
11:    reduceFanout(newBucket);
12:  else
13:    self.updatePQ(); /* update this node's entry in the priority queue /*
14:  end if
15:  if root.cntBuckets > b_max then
16:    root.reduceBuckets();
17:  end if
18: end if
```

Before we discuss how to enforce the two constraints and how to use the priority queue for this purpose, we need to pay attention to another issue. Because bounding boxes are allowed to overlap, it is possible that there are several child nodes whose MBBs contain p's coordinates (line 1). In such cases we have to decide which of these child nodes $M = \{N_1, N_2, \ldots, N_n\} \subseteq childNodes$ would be the best choice, i.e., the most responsible for p. We define that node to be most responsible whose bounding box center has the least distance to p. Hence, for each $N \in M$ we compute a responsibility coefficient c_r according to Equation 5.1 and choose that N that minimizes c_r. $N.low[i]$ and $N.high[i]$ ($N.low[i] \leq N.high[i]$) represent the lower and upper boundaries of node N's bounding box in dimension i. c_r represents the distance of p to the center of the MBB normalized

by the MBB's extension. If there are multiple nodes with the same minimum coefficient, we randomly choose one of them. In case the bounding box has no extension in dimension i (i.e., $N.low[i] = N.high[i]$), dimension i does not have any impact on the value of c_r.

$$c_r(N,p) := \sum_{\substack{d \in \{1,...,d_{max}\} \\ N.low[d] \neq N.high[d]}} \frac{|\frac{1}{2}(N.low[d] + N.high[d]) - p[d]|}{N.high[d] - N.low[d]} \tag{5.1}$$

Note that buckets do not necessarily need to have extensions in all dimensions. It is, for example, even possible that a bucket has no extension at all. This happens when we insert new records as buckets. For such a bucket the responsibility coefficient c_r would be 0.

The reason why we allow regions to overlap is simple. In order to avoid the overlap, we would have to split existing buckets and reassign the contained records to the new buckets. As we do not keep detailed information about already inserted data records, we cannot split existing regions and rearrange the membership of already inserted data records. In order to split regions nonetheless, we would have to assume a uniform data distribution of records within a bucket. This would lead to errors in the estimation of result cardinality. In order to keep these estimation errors as minimal as possible, we try to find an existing bucket whose MBB contains the coordinates of the new data record p before calling method *insertDataRecord*. If we did not do so, it would be possible that, despite following Algorithm 4, we would not find an existing bucket that contains p even though it might exist.

Another consequence of summarizing records is that we cannot easily balance the QTree by applying the same or similar strategies that are used for the R-tree. The reasons are the same as stated above: splitting buckets cannot be realized without approximation errors, which should be avoided. Nevertheless, we can apply several heuristics that still prevent the QTree from degenerating into a linear list. We discuss them in the following.

Reducing the Number of Buckets

Whenever the number of buckets exceeds b_{max}, method *reduceBuckets* - sketched in Algorithm 5 - is called to merge some of the QTree's buckets. This might be necessary after having inserted a new data record or when the amount of space the QTree is allowed to allocate is reduced during runtime.

We apply a greedy strategy and always choose those two sibling buckets (of the whole QTree) for merging that minimize the penalty function defined by Equation 5.2. For two buckets (B_1,B_2) τ denotes the penalty for merging them.

$$P_{M_{Bucket}} : (B_1, B_2) \mapsto \tau \qquad B_1, B_2 : \text{Buckets},\ 0 < \tau \in \mathbb{R} \tag{5.2}$$

A naïve strategy to find the best pair of buckets would be to compute the penalty function for each possible pair of sibling buckets and merge the pair that minimizes τ. As this is a very expensive solution to the problem, a QTree maintains a priority queue.[1]

[1] This is what gave the QTree its name: we named the tree structure after the priority queue that we use for its implementation.

Its entries are key-value pairs sorted in ascending key order: $key = P_{M_{Bucket}}(B_1, B_2)$, $value = parent.getReference()$ with B_1 and B_2 being children of node *parent*. Thus, the priority queue provides links to nodes (*value*) with a pair of children whose mergence would cause a certain penalty (*key*). As the priority queue only contains links to the parent node, the node itself keeps links to the pair of its child nodes that minimizes the penalty function. As we only consider this pair, each node has at most one entry in the priority queue.

The penalty function $P_{M_{Bucket}}$, as introduced above, is a measure for the approximation error caused by merging buckets. We determine the penalty based on the MBB that the bucket resulting from the mergence would have. Since we want to use the QTree as base structure for routing indexes, we are interested in "small" bounding boxes. Thus, we use the penalty function defined by Equation 5.3, which is defined on the maximum extension in any dimension of the MBB resulting from the mergence. This leads to buckets that have a similar extension in all dimensions. Such buckets are well-suited for routing indexes as we will see later. As dimensions/attributes often have different ranges of values, for example the values of one dimension could be restricted to the interval $[0, 10]$ whereas another dimension could be restricted to $[0, 1000]$, Equation 5.3 uses an appropriate normalization based on these intervals denoted as $[DimSpec[d].low, DimSpec[d].high]$ for dimension d.

$$P_{MaxBound}(B_1, B_2) := \max_{d \in \{1, ..., d_{max}\}} \left\{ \frac{\max\{B_1.high[d], B_2.high[d]\} - \min\{B_1.low[d], B_2.low[d]\}}{scale(d)} \right\} \tag{5.3}$$

$$scale(d) := \begin{cases} DimSpec[d].high - DimSpec[d].low, & \text{if } DimSpec[d].low \neq DimSpec[d].high \\ 1, & \text{otherwise} \end{cases}$$

Note that even if dimensions cannot be specified by minimum and maximum values in advance, we can still use the measure defined in Equation 5.3 by adapting minimum and maximum values per dimension based on the attribute values of inserted data records. If the interval changes in a considerable manner (e.g., the growth exceeds a predefined threshold), we can still recompute the penalties for the priority queue.

Let us get back to method *reduceBuckets* and Algorithm 5. At first, the method is called at the root node of the QTree (lines 1–5). As it is in principle possible that b_{max} is not only exceeded by 1 but by n, we reduce the number of buckets until b_{max} is no longer exceeded. Each time we need to obtain the node with the least merge penalty from the priority queue. Having found such a node (*node*), method *reduceBuckets* is called at that node. Note that Algorithm 5 only sketches the algorithm and does not consider all special cases such as a tree only consisting of the root node and several buckets.

If the node has only two child nodes (buckets), it itself is converted into a bucket that subsumes the former two (line 7–11). This step includes (i) merging the two buckets by combining their statistics, (ii) determining an MBB that comprises those of the two buckets, (iii) updating the QTree's global bucket counter, (iv) removing the node's entry from the priority queue, (v) replacing the node with the newly created bucket, and finally (vi) updating the priority queue with respect to the entry of its parent node (line 9). Afterwards, we can exit method *reduceBuckets* since two buckets have been

Algorithm 5 *reduceBuckets()*

```
1: if self is root then
2:    while root.cntBuckets > b_max do
3:       node = priorityQueue.pop();
4:       node.reduceBuckets();
5:    end while
6: else
7:    if #self.childNodes = 2 then
8:       self.convertToBucket();
9:       parent.updatePQ();
10:      return;
11:   end if
12:   (c_1,c_2) = self.getLocalNextToMerge();
13:   mergedBucket = createNewBucket(c_1,c_2);
14:   self.childNodes = self.childNodes \ {c_1, c_2} ∪ {mergedBucket};
15:   destroyed = self.tryToRemoveNode();
16:   if destroyed = false then
17:      self.updatePQ();
18:   end if
19: end if
```

merged successfully.

If the node has more than two child nodes, we have to consider the two children (c_1,c_2) whose penalty is the lowest (line 12). c_1 and c_2 are then merged into a single bucket by creating a new bucket node whose statistics and MBB emerge from those of c_1 and c_2 (line 13). Subsequently, c_1 and c_2 are removed as children and the new merged bucket is added as a child (line 14). Since the node has now less children than before, we try to remove it (line 15). Only if this is impossible, the priority queue has to be updated with respect to the parent node (as there was a change in its child buckets).

Algorithm 6 *tryToRemoveNode()*

```
1: if self is root or self is bucket then
2:    return false;
3: end if
4: if #parent.children + #self.children − 1 ≤ f_max then
5:    parent.addToChildren(self.children);
6:    parent.removeFromChildren(self);
7:    removeFromPQ(self);
8:    parent.updatePQ();
9:    destroy(self);
10:   return true;
11: else
12:   return false;
13: end if
```

Method *tryToRemoveNode* (Algorithm 6) removes an inner node if all its children can be added to its parent node without violating the fanout constraint. Hence, if this constraint is not violated (line 4), all children are removed and added as children to the parent node (line 5). Of course, this is not possible if the method is called at the root node of the QTree (lines 1–3). Before finally destroying the node (line 9), its entry has to be removed from the priority queue (line 7) and the entry of its parent node has to be updated (lines 8).

Reducing the Fanout of Inner Nodes

As we insert a data record p with a new bucket whenever we cannot find a bucket with an MBB that contains p's coordinates, it is possible that after the insertion the number

of children of an inner node exceeds f_{max}. In that case, we need to merge some of its children using method *reduceFanout* (Algorithm 7).

In order to reduce the fanout of a node $self$, we need to find the pair of sibling child nodes (not only buckets) (c_1, c_2) that yields the minimum penalty (line 1). A penalty function we can use in this context is defined by Equation 5.4.

$$P_{M_{Node}} : (N_1, N_2) \mapsto \tau \qquad N_1, N_2 : \text{Buckets} \cup \text{Inner Nodes},\ 0 < \tau \in \mathbb{R} \tag{5.4}$$

As this is only a generalized version of Equation 5.2 we can use Equation 5.3 to compute the penalty. Thus, we compute the penalty for each pair of $self$'s children disregarding whether the children are buckets or inner nodes. Since this method is called only when a new bucket ($newBucket$) has been inserted, we have a reference to that bucket as input and distinguish two situations: (i) $newBucket \in \{c_1, c_2\}$ and (ii) $newBucket \notin \{c_1, c_2\}$.

Algorithm 7 *reduceFanout(newBucket)*

```
1: (c1, c2) = getChildrenMinimizingPenalty();
2: if newBucket ∈ {c1, c2} then
3:   c_base = {c1, c2} \ newBucket;
4:   if c_base is an inner node then
5:     self.childNodes = self.childNodes \ newBucket;
6:     c_base.addToChildren(newBucket);
7:     c_base.adaptMBB();
8:     if #c_base.childNodes > f_max then
9:       c_base.reduceFanout(newBucket);
10:    else
11:      c_base.updatePQ();
12:    end if
13:    return;
14:  end if
15: end if
16: newInnerNode = createNewInnerNode(c1,c2);
17: self.childNodes = self.childNodes \ {c1, c2} ∪ {newInnerNode};
18: destroyed = false;
19: for n ∈ newInnerNode.childNodes do
20:   destroyed = destroyed OR n.tryToRemoveNode();
21: end for
22: if destroyed = false then
23:   newInnerNode.updatePQ();
24: end if
25: self.updatePQ();
```

The former case demands special attention (lines 2–15): if $newBucket \in \{c_1, c_2\}$, then let c_{base} denote the node that has been chosen to be merged with $newBucket$ (line 3). In case c_{base} is not a bucket, i.e., it has further child nodes, $newBucket$ is removed as child from the current node (*self*) and attached as a child to c_{base} (lines 5–6) – this may cause the adaption of c_{base}'s MBB (line 7). If this results in a situation where c_{base} now has more than f_{max} children, method *reduceFanout* is recursively called at node c_{base} (lines 8–9). Otherwise, if c_{base} does not have more than f_{max} children (lines 10–12), its entry in the priority queue has to be updated by explicitly calling method *updatePQ*. This is necessary because c_{base} now has a new child node whose mergence with one of the other children could result in a lower penalty value than the mergence of any two of the old ones. If $newBucket \in \{c_1, c_2\}$ and if c_{base} is no bucket, then method *reduceFanout* ends at this point because $newBucket$ has been inserted successfully.

In all other cases, we have to create a new inner node ($newInnerNode$) with c_1 and c_2 as children (line 16-17). In order to prevent the tree structure from degenerating

into a list, we try to remove $newInnerNode$'s child nodes (lines 19–21) by applying the heuristic sketched in Algorithm 6. If none of these child nodes could be removed, we need to explicitly update the priority queue entry for $newInnerNode$ (lines 22-24). Finally, $self$'s entry in the priority queue has to be updated since there have been changes to its children (line 25).

Note that this approach has another effect: we obtain a QTree whose nodes have as many children as possible but at most f_{max}. As we only consider sibling buckets for merging, having as many children as possible permits our algorithm to find a good pair of child buckets that minimizes $P_{M_{Bucket}}$. Furthermore, the tree has less levels and less inner nodes, which reduces not only lookup time but also memory consumption.

Inserting Buckets and Merging QTrees

In order to construct routing indexes, it is necessary to combine the information of their base structures. As we consider QTrees as base structures, we need to find a way to merge the information of QTrees. This can be realized by simply inserting the information of one QTree into the another. For this purpose, we only need to pay attention to the buckets of the tree whose information we want to insert because inner nodes do not provide any information about represented data. However, so far we have only discussed how to insert data records into a QTree, let us now briefly discuss how to insert buckets.

For this purpose, we can use the same principles that we use to insert data records. At first, we look for a bucket in the tree whose MBB is large enough to completely enclose the MBB of the bucket we want to insert (B_{insert}). If we can find such a bucket, we update its statistics (by adding the number of records contained in B_{insert}) and have successfully inserted B_{insert} into the tree.

If we cannot find such a bucket, we call a slightly modified version of method *insert-DataRecord* (Algorithm 4). We need to determine when a node is responsible for B_{insert}. This is the case when it completely encloses B_{insert}'s MBB. In case there is more than just one such node, we use the center coordinates of B_{insert}'s MBB as p in Equation 5.1. According to the responsibility coefficient, we can now decide which node is responsible for B_{insert} and recursively call the method at that node. If we do not find such a responsible node, we insert B_{insert} as a new child and proceed as indicated in lines 4 through 20 (except that we do not have to create a new bucket).

5.2.3 Lookups

After having introduced how to construct QTrees, let us briefly review how they can be used to determine whether a peer holds relevant data to a particular query or not. At this point, we only focus on range queries since we discuss the usage of DDSs for skyline and top-N queries later in Chapter 6. Remember that the reason for applying routing indexes is to identify neighbors that do not provide relevant data to a query. Furthermore, false negative routing decisions, i.e., a neighbor is classified as irrelevant although it provides relevant data, should be avoided at any cost. False positive routing decisions in contrast, can be accepted although their number should be minimized as they unnecessarily increase query execution costs.

With respect to QTree-based distributed data summaries, identifying relevant neighbors with respect to a particular query means to compare the constraints defined by the query (e.g., $attribute1 < 20$, $attribute2 > 10$, etc.) to the buckets' MBBs. If the region defined by the constraints overlaps the MBB of any bucket within the QTree, the corresponding neighbor is classified as relevant to the query and will be considered by the query processing strategy. Thus, the following steps sketch the corresponding algorithm:

1. parse the query expression and extract constraints,
2. start at the root node of the QTree and identify all child nodes whose MBBs support the constraints, i.e., those that do not contradict,
3. proceed recursively with the child nodes until we arrive at the bucket level or no more relevant children can be identified, and
4. once we have found a bucket whose MBB supports the constraints, the corresponding neighbor is classified as relevant since the index describes data that should be considered for processing the query.

An advanced version of this algorithm can also determine the expected number of results by taking all buckets into account that support the constraints. For all these buckets we simply need to determine the degree of overlap between their MBBs and the region defined by the constraints. Assuming a uniform distribution of records within an MBB, the size of the overlapping region multiplied by the factor $\frac{\text{statistical entry}}{sizeOf(MBB)}$ describes the number of records that are provided by the neighbor with respect to a particular bucket. This estimation can be improved by adding additional information to a bucket that describes the distribution of records within its MBB.

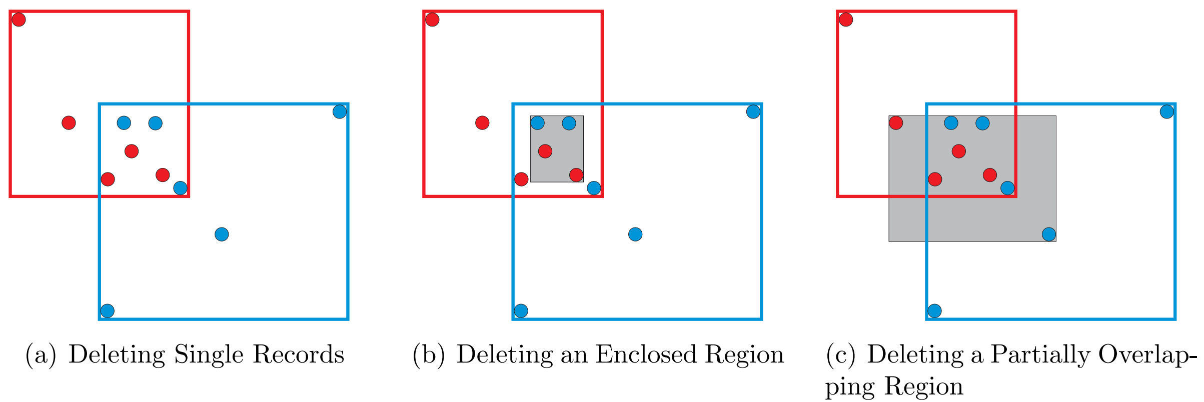
5.2.4 Deletions

We have already discussed how to insert records into a QTree (Section 5.2.2). In addition, we need to consider how to remove records. As we will see later, it might be possible that we do not receive detailed information about the deletion of single data records but only the information that in a specific subspace of the data space a certain number of records has been deleted. Although we are only discussing this kind of deletions in the following, the same algorithms can be applied to the deletion of individual records as they can also be treated as regions.

As input to the algorithm we receive a region B_d and a number B_d.count – meaning that in region B_d B_d.count records are deleted. In principle, the algorithm runs in three steps:

1. find a responsible bucket,
2. decrement its statistical entry, and
3. propagate changes within the tree.

At first, we need to find a bucket that represents B_d. In the simplest case, we compare B_d to all buckets and find a bucket B that completely encloses B_d. Once we have determined such a bucket, its statistical entry is reduced by B_d.count. If the count value is reduced to 0, B is removed and heuristics are applied to balance the tree and to minimize the



(a) Deleting Single Records (b) Deleting an Enclosed Region (c) Deleting a Partially Overlapping Region

Figure 5.4: Deletions in QTrees with Overlapping Buckets

MBBs of the nodes on the way from B_d to the root – these MBBs might become smaller as they do no longer represent the MBB of the deleted bucket.

However, as Figure 5.4 illustrates, deletion is not always that straightforward. The problem is that regions and thus buckets are allowed to overlap. In that case it is much harder to determine a responsible bucket. Figure 5.4(a) shows two buckets and the records they represent. If we now want to delete any of the records contained in the overlapping region, we cannot know due to the approximation into which of the two buckets we have inserted them in the first place. The same problem is illustrated in Figure 5.4(b). The difference is that now we do not want to delete single records but a region that is completely contained in two buckets. Thus, it is hard to determine which bucket is responsible. Finally, Figure 5.4(c) shows another problem we need to consider: we want to delete a region that is not completely contained in any bucket but partially overlaps several buckets.

As deleting a single record and deleting a region can be reduced to the same problem, we only need to distinguish between two cases: (i) there is only one bucket that intersects the region defined by B_d and (ii) there are several such buckets. To solve this problem, we consider three strategies.

Strategy 1: Merging Overlapping Buckets

Algorithm 8 sketches one strategy that aims at solving the problem of deleting a region B_d when we have multiple buckets that at least partially overlap. In the first step, we need to find the set ($respBuckets$) of all buckets that at least partially overlap B_d (line 1). Thus, $respBuckets$ contains all buckets that completely enclose B_d as well as all those buckets that only partially overlap B_d. Next, we have to distinguish whether we have found only one such bucket or multiple buckets. Note that if we cannot find such a bucket at all, this means that something shall be deleted from the tree that has never been inserted. Algorithm 8 does not go into details on how to handle such special cases.

If there is only one bucket that at least partially overlaps B_d (lines 2–3), method *removeFromBucket* removes B_d from that bucket by decrementing its statistical entry by the number of records to delete in B_d. If there are multiple buckets that overlap B_d

Algorithm 8 *deleteRegionMerge(B_d)*

```
1: respBuckets = getAllOverlappingBuckets(B_d);
2: if respBuckets.size() = 1 then
3:   removeFromBucket(respBuckets[0], B_d);
4: else
5:   if respBuckets.size() > 1 then
6:     mergeBucket = new Bucket();
7:     for b ∈ respBuckets do
8:       removeFromBucket(b,b);
9:       mergeBucket.enlargeBounds(b);
10:      mergeBucket.count += b.count;
11:    end for
12:    root.insertDataRegion(B_d);
13:    removeFromBucket(mergeBucket,B_d);
14:  end if
15: end if
```

(lines 4–14), we first merge them all into one bucket (*mergeBucket*) and remove them from the tree. Once we have inserted *mergeBucket* into the tree (line 12), we can delete B_d from *mergeBucket* without committing any approximation errors (line 13) because it represents all records within B_d's MBB that have ever been inserted into the tree.

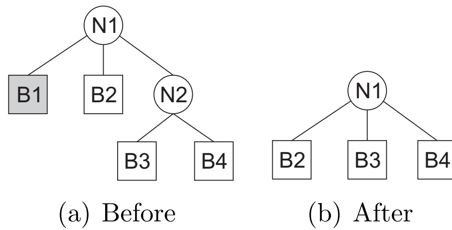
Method *removeFromBucket* (Algorithm 9) takes care of "deleting entries" from a bucket. The algorithm we present here does not show every detail such as ensuring that the two buckets that are given as parameters are compatible to each other. However, we have to distinguish whether the count of the bucket B, which the data is to be deleted from, is greater than the count of B_d. If B.count is greater, then B.count is decremented by B_d.count (lines 15–17).

Algorithm 9 *removeFromBucket(Bucket B, BucketToDelete B_d)*

```
1: if B.count ≤ B_d.count then
2:   removeBucketAsChildFromParent(B);
3:   decrementGlobalBucketCounter();
4:   B.parent.adaptMBBPropagateBottumUp();
5:   removed = false;
6:   /* case 1 */
7:   for QTreeNode n ∈ B.getFormerSiblings() do
8:     removed = removed OR n.tryToRemoveNode();
9:   end for
10:  /* case 2 */
11:  removed = removed or B.parent.tryToRemoveNode();
12:  if removed = false then
13:    B.parent.updatePQ();
14:  end if
15: else
16:  B.count -= B_d.count;
17: end if
```

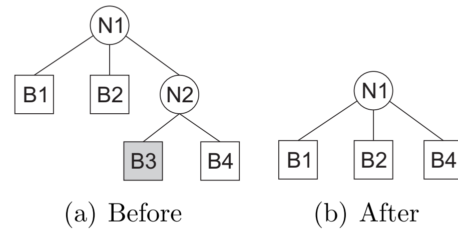
In any other case (B.count $\leq B_d$.count), we need to remove bucket B from the tree (lines 2–14) because after the deletion it is empty. At first, B is detached, i.e., removed from the list of children of its parent (line 2), the number of buckets in the QTree is decremented by 1 (line 3), and the MBB of its former parent node as well as the MBBs of all nodes on the way from the parent to the root node are adapted (line 4) – as it is possible that the MBBs may now be reduced in size. Finally, we apply some heuristics that prevent the tree from degenerating into a linear list. For this purpose, we call method *tryToRemoveNode* (Algorithm 6) at each of B's former siblings (lines 7–9) and at its former parent node (line 11). As that method takes care of updating the priority

queue if any node is removed, we need to update the priority queue explicitly only if no node has been removed (lines 12-14).



(a) Before (b) After

Figure 5.5: Deletion of Buckets - Removing a Sibling Node



(a) Before (b) After

Figure 5.6: Deletion of Buckets - Removing the Parent Node

Figures 5.5 and 5.6 illustrate the two cases of possible degeneration that we have to counteract. Assume we have a QTree with $f_{max} = 3$. Figure 5.5 illustrates why we have to call method *tryToRemoveNode* at each sibling of the deleted bucket B_1. By calling that method at N_2, N_2 is destroyed and its child nodes (buckets) are added as children of N_1. After the removal of B_1, this is possible without violating the f_{max} constraint. As a result the height of the QTree has been reduced by one.

Figure 5.6 illustrates the second case that we have to consider. In this example bucket B_3 is deleted. If we did not try to rebalance the tree, N_2 would only have one child node, i.e., there is no need to keep N_2. Thus, by calling method *tryToRemoveNode* at B_3's parent we can remove N_2 and again obtain a QTree with decreased height.

Strategy 2: Minimum Distance

This variant, sketched in Algorithm 10, deletes records from buckets in dependence on their distance to B_d. At first, we again need to find all buckets that at least partially overlap the region defined by B_d (line 1). If there exists at least one such bucket, the list of buckets (*respBuckts*) is sorted in ascending order by their distance to B_d (line 3) – applying Equation 5.1 and the two center coordinates of the bucket and B_d. Next, the algorithm iteratively runs through the list of overlapping buckets starting at the bucket with the minimum distance to B_d. For each bucket we test whether its statistical entry is higher than the number of records that still need to be removed (line 6). If it is higher (lines 10–13), we simply decrease the bucket's statistical entry accordingly and exit the method. If the bucket has a statistical entry smaller than or equal to the number of records that still need to be deleted (lines 6–9), the bucket is removed and the algorithm proceeds with other buckets until all $B_d.count$ records are deleted.

As Figure 5.7 illustrates, it is possible that when applying this strategy records are deleted from the "wrong" bucket. Assume we intend to delete the white records within the highlighted region. As B1's center is closest to the center of the highlighted region, B1 would be removed although the original records have not been inserted into it. This is impossible for the first strategy where all candidate buckets are merged into a larger bucket that the records are deleted from.

However, when we later on want to delete records from a bucket that no longer exists, we need to determine some other buckets to effectuate the deletion. Thus, we restart the

Algorithm 10 *deleteRegionMinimumDistance(B_d)*

```
1: respBuckets = getAllOverlappingBuckets(B_d);
2: if respBuckets.size() ≥ 1 then
3:   sortByDistance(respBuckets,B_d);
4:   i = 0;
5:   while B_d.count > 0 and i < respBuckets.size() do
6:     if B_d.count ≥ respBuckets[i].count then
7:       count = respBuckets[i].count;
8:       removeFromBucket(respBuckets[i], respBuckets[i]);
9:       B_d.count -= count;
10:    else
11:      removeFromBucket(respBuckets[i],B_d);
12:      B_d.count = 0;
13:    end if
14:    i = i + 1;
15:  end while
16: end if
```

algorithm and adapt line 1: *respBuckets* now is the set of all buckets that exist in the QTree. By sorting them according to their distance to B_d and deleting the remaining records from regions close-by, we have a good chance to compensate the mistakes made by previous deletions.

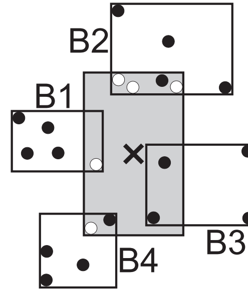


Figure 5.7: Problem for Deletion Strategy 2: 5 *records (white) are to be deleted from the highlighted region. Applying deletion strategy* 2*,* B_1 *is wrongly decremented by* 5 *and thus removed.*

Strategy 3: Percentage of Overlap

A third strategy we can use to remove data from the QTree is sketched in Algorithm 11. The principle is to adapt the buckets' statistical entries in dependence on their degree of overlap with B_d. This means that the more a bucket overlaps B_d, the higher is the decrease of its statistical entry.

At first, we determine the set of buckets that at least partially overlap B_d (line 1). If there is at least one such bucket, we need to compute the overlap (volume) between each of these buckets b and B_d. Then, B_d.count is split up according Equation 5.5 (line 3):

$$d(b) = \frac{B_d.\text{count}}{\sum_{b \in respBuckets}(volume(b, B_d))} \times volume(b, B_d) \tag{5.5}$$

Thus, we can determine a specific number of deletions for each bucket individually in dependence on the degree of overlap with B_d. It might be possible that $d(b)$ is higher than b.count. In that case the algorithm reduces the assigned deletion value to b.count

Algorithm 11 *deleteRegionPercentageOverlap(B_d)*

```
1: respBuckets = getAllOverlappingBuckets(B_d);
2: if respBuckets.size() > 0 then
3:    computeOverlapDeletionPerBucket(respBuckets);
4:    for b ∈ respBuckets do
5:       if deletion(b) > b.count then
6:          deletion(b) = b.count;
7:       end if
8:    end for
9:    for b ∈ respBuckets do
10:      removeFromBucket(b,new Region(B_d.MBB,deletion(b)));
11:      B_d.count -= deletion(b);
12:   end for
13:   if B_d.count > 0 then
14:      deleteRegionPercentageOverlap(B_d);
15:   end if
16: else
17:   deleteRegionMinimumDistance(B_d);
18: end if
```

(lines 4–8). Then, the deletion of $d(b)$ is performed at each bucket (lines 9–12). In that process, buckets are removed from the tree if their statistical entry is reduced to 0. It is possible that for several buckets b not all $d(b)$ records could be deleted (if b.count $< d(b)$). Thus, not all B_d.count records could have been deleted. In that case (lines 13–15), the remaining deletion value is distributed among the remaining overlapping buckets by recursively invoking Algorithm 11. However, we might still encounter the same problem that we had with strategy 2: there might be no buckets that overlap B_d's MBB. Thus, Algorithm 10 is called to solve this problem (lines 16–18).

Another weakness of this strategy is that updating just one record might cause changes in multiple buckets. Applying Equation 5.5 it is possible that the statistical value of a bucket is reduced by only 0.1111. With respect to query processing we need to decide what a bucket with a count below 1 means. Does it represent a record or does it not? In order to avoid missing relevant data to a query, the query processing strategy should pay attention to such buckets.

The most important problem is that we might delete records from buckets into which the records have never been inserted. This is the same problem that we have already encountered in strategy 2. It means that the information represented by the routing indexes cannot be completely relied on by the query processing strategy (false negative routing decisions). Thus, from the query processing strategy's point of view the best solution to the deletion problem is strategy 1, where this problem cannot occur.

We are aware that the basic strategies for deletion can be improved. Alternative strategies could introduce buckets with negative statistical values, for example realized by two QTrees (one representing insertions, one representing deletions). This would shift the decision which records can assumed to be deleted to the query processing strategy. Although this might be an interesting strategy, discussing and elaborating it would go beyond the scope of this dissertation. Thus, in the following we apply the first strategy if not explicitly stated otherwise.

5.2.5 Penalty Functions

In Section 5.2.2 we have defined a penalty function $P_{M_{Bucket}}$ (Equation 5.2) to determine the penalty for merging two buckets. As instance of this definition we have presented a penalty function defined on the maximum extension of a bucket's dimensions (Equation 5.3):

$$P_{MaxBound}(B_1, B_2) := \max_{d \in \{1,...,d_{max}\}} \left\{ \frac{\max\{B_1.high[d], B_2.high[d]\} - \min\{B_1.low[d], B_2.low[d]\}}{scale(d)} \right\}$$

$$scale(d) := \begin{cases} DimSpec[d].high - DimSpec[d].low, & \text{if } DimSpec[d].low \neq DimSpec[d].high \\ 1, & \text{otherwise} \end{cases}$$

In this section we discuss further variants of penalty functions. In order to compare them, let us introduce the example data set illustrated in Figure 5.8. It consists of 1000 two-dimensional data records, each attribute value, i.e., each dimension, is restricted to the interval $[0, 1000]$. The attribute values of each record are chosen randomly in the interval. The data records are inserted in random order into a QTree with $f_{max} = 4$ and $b_{max} = 150$. However, the order of insertion is the same for all tests discussed in this section. The buckets we obtain by applying $P_{MaxBound}$ are illustrated in Figure 5.9. They have a relatively equal extension in all dimensions, which is – as we will see later – most beneficial for routing indexes.

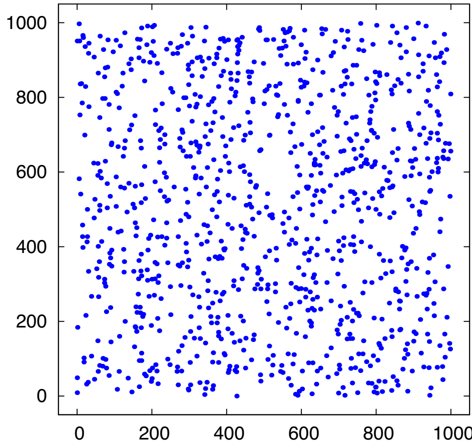


Figure 5.8: Random Test Data

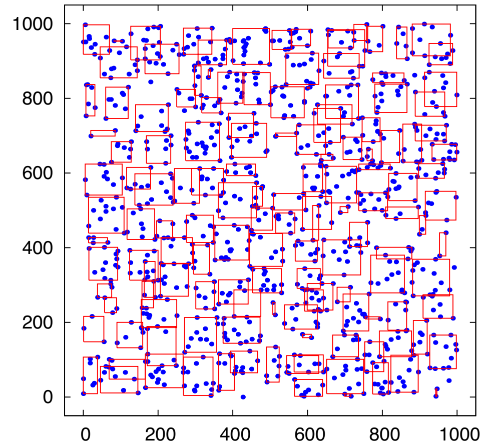


Figure 5.9: Random Test Data in a QTree Using $P_{MaxBound}$ as Penalty Function

At first, let us consider some variations defined on the sum of a buckets' extensions in all dimensions (Equation 5.6) and the average of these extensions (Equation 5.7).

$$P_{SumDim}(B_1, B_2) := \sum_{d=1}^{d_{max}} \frac{\max\{B_1.high[d], B_2.high[d]\} - \min\{B_1.low[d], B_2.low[d]\}}{scale} \tag{5.6}$$

$$P_{AvgDim}(B_1, B_2) := \frac{1}{d} \sum_{d=1}^{d_{max}} \frac{\max\{B_1.high[d], B_2.high[d]\} - \min\{B_1.low[d], B_2.low[d]\}}{scale} \tag{5.7}$$

Just as similar as Equations 5.6 and 5.7 look like are the results that we obtain by applying them. Thus, Figures 5.10 and 5.11 look very similar. In comparison to Figure 5.9 we can see that the buckets created when applying these two penalty functions have a slightly higher tendency to vary in their dimensional extensions.

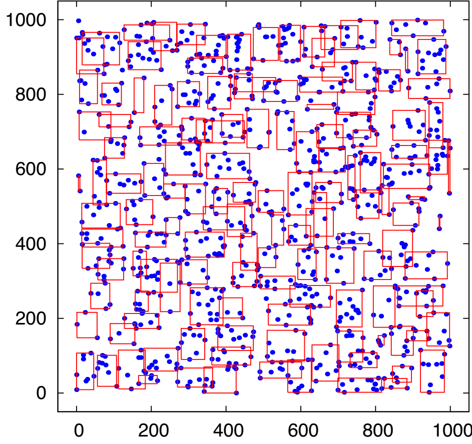


Figure 5.10: Random Test Data in a QTree Using P_{SumDim} as Penalty Function

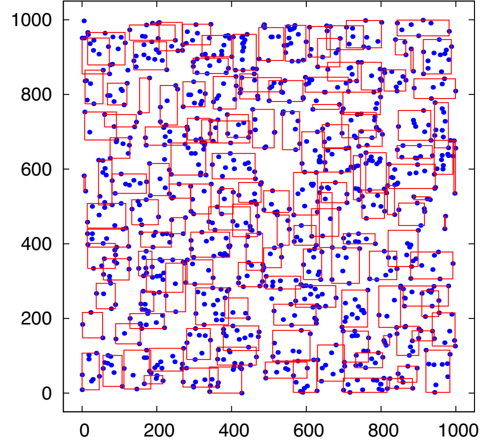


Figure 5.11: Random Test Data in a QTree Using P_{AvgDim} as Penalty Function

Another measure we can use as penalty function is based on the diameter of a bucket. By applying the Euclidean Distance and a normalization by the dimensions' attribute ranges, we obtain:

$$P_{Diameter}(B_1, B_2) := \sqrt{\sum_{d=1}^{d_{max}} \left(\frac{\max\{B_1.high[d], B_2.high[d]\} - \min\{B_1.low[d], B_2.low[d]\}}{scale} \right)^2} \tag{5.8}$$

Figure 5.12 illustrates the QTree buckets we obtain by applying $P_{Diameter}$ on the QTree test data of Figure 5.8. The resulting buckets are similar in size and shape to the ones that we obtain by applying P_{MaxDim}. However, there is still a small tendency to create buckets with a higher extension in some of the dimensions.

We can also define measures based on the volume of buckets. Equation 5.9 defines a corresponding measure.

$$P_{Volume}(B_1, B_2) := \prod_{\substack{i=1 \\ DimSpec[d].high \neq DimSpec[d].low}}^{d} \left(\max\{B_1.high[d], B_2.high[d]\} - \min\{B_1.low[d], B_2.low[d]\}\right) \tag{5.9}$$

The problem we have to solve is that buckets do not necessarily need to have an extension in each dimension (e.g., buckets representing one single record). Thus in order to avoid the multiplication by 0, P_{Volume} only considers dimensions with extensions. However, in case a bucket has no extension in any dimension, the volume is 0 (not expressed by Equation 5.9). Figure 5.13 shows the result we obtain by applying P_{Volume} on our example data. There are many buckets with a large extension in one dimension and a short extension in other dimensions. With respect to query routing, we should avoid such buckets. Assume we have a range query, then the chance that such a "long" bucket intersects the queried range is higher than for a bucket with relatively equal extensions in all dimensions. As the routing strategy would have to consider peers providing such data, we should avoid such buckets.

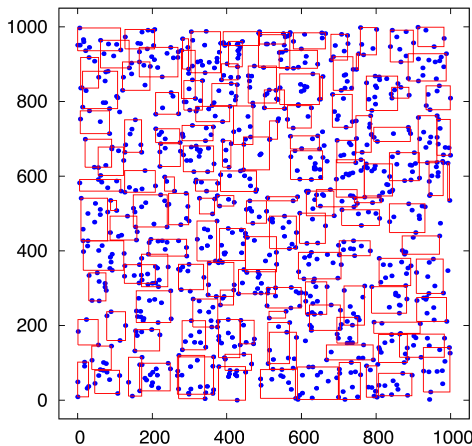


Figure 5.12: Random Test Data in a QTree Using $P_{Diameter}$ as Penalty Function

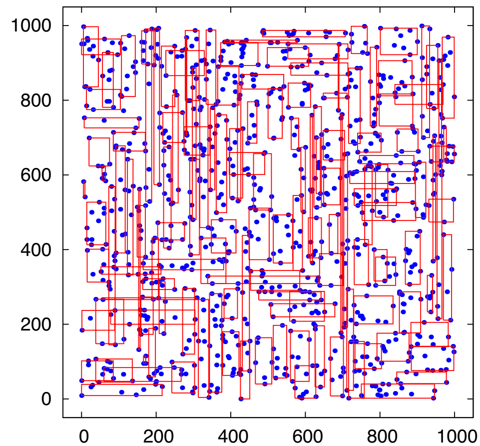


Figure 5.13: Random Test Data in a QTree Using P_{Volume} as Penalty Function

In future work, more sophisticated penalty functions could be developed that take other aspects of buckets into account, e.g., how their MBBs overlap. Furthermore, it is an interesting approach to use different penalty functions for merging buckets (when reducing the number of buckets) and for merging inner nodes (when reducing the fanout of an inner node).

Comparison of Penalty Functions

We have presented several variants of penalty functions that we can use to build a QTree. So far, we have only illustrated how the resulting buckets look like. This only shows on an intuitive basis what penalty functions construct small buckets. In order to compare the penalty functions on an objective basis, we introduce the "average bucket density" (Equation 5.10) as an appropriate measure. It is defined on the ratio between the number of records contained in a bucket and its size. The higher the density, the better is the representation.

$$Density(B) = \underset{B \in QTree}{\text{avg}} \left\{ \frac{B.count}{\prod_{i=1}^{d} (B.high[i] - B.low[i] + 1)} \right\} \tag{5.10}$$

Figure 5.14 shows the results for the application of the penalty functions discussed above on the random data set. It shows how the average density of all buckets changes when more and more data is inserted into the QTree. For each test run we inserted the data records in the same random order ($f_{max} = 4$, $b_{max} = 150$). Note that the average bucket density in Figure 5.14 is displayed in logscale.

Our results with respect to inserting the random data of Figure 5.8 show that until we we have inserted 150 records into the tree, the bucket density is 1, which means each bucket is filled optimally. This is not astonishing because each of those 150 records is represented by its own bucket (a bucket without extension). When we insert the 151st record, for the first time two records need to be represented by the same bucket. Consequently, the average bucket density decreases for all penalty functions.

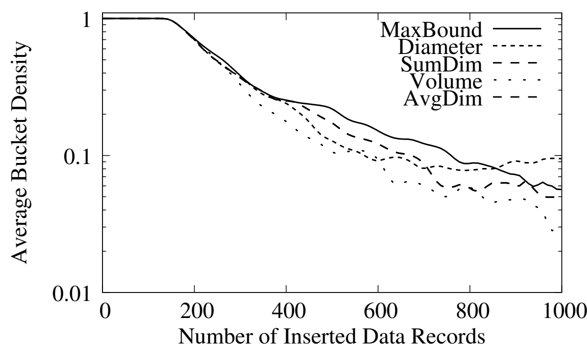


Figure 5.14: Comparison of Penalty Functions – Random Data

P_{Volume} results in the worst results because the buckets cover rather large regions that do not contain records in the original data set. As we have already anticipated from the buckets shown in Figures 5.10 and 5.11 as well as from Equations 5.6 and 5.7, penalty functions P_{SumDim} and P_{AvgDim} produce buckets that are similar if not even the same. In most cases $P_{MaxBound}$ produces the best results in comparison to all the other penalty functions. However, with large numbers of inserted records $P_{Diameter}$ performs actually better than $P_{MaxBound}$.

Of course, we also tested other data distributions such as clustered data. The example data set that we used as a representative for our tests is illustrated in Figures 5.15. It is restricted to the same data space as the random test data of Figure 5.8, i.e., all attribute values are restricted to the interval $[0, 1000]$. The clustered data set consists of 100 clusters with 10 data records each (thus 1000 in total). The attribute values of a record have a random Euclidean Distance to the cluster center of at most 30.

For this data set we ran the same test runs with the same QTree parameters ($f_{max} = 4$ and $b_{max} = 150$). For each test run the records were again inserted in the same random order. The results we obtained for the clustered data set are shown in Figure 5.16. For this data set we see that in general $P_{MaxBound}$ creates QTrees with the best average

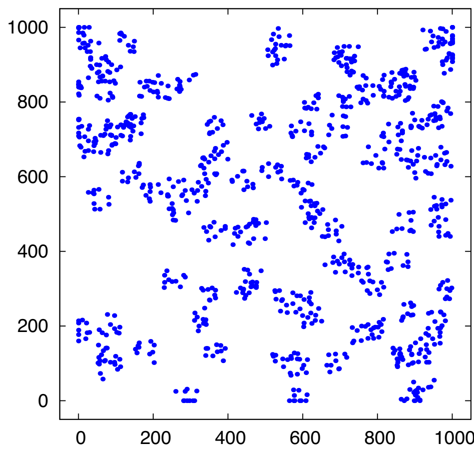


Figure 5.15: Cluster Test Data

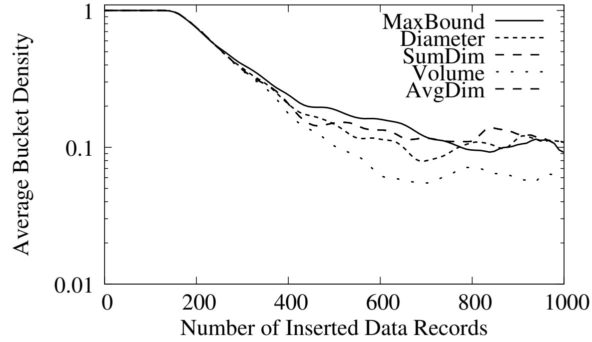


Figure 5.16: Comparison of Penalty Functions – Clustered Data

bucket density. P_{SumDim} and P_{AvgDim} again create QTrees with the same bucket densities, their results are sometimes even a little better than the results of all the other penalty functions. In contrast to the random data set, P_{Volume} performs better but nevertheless $P_{MaxBound}$ and $P_{Diameter}$ provide better results.

In summary, $P_{MaxBound}$ yields the best results for both tested data sets. Although P_{SumDim}/P_{AvgDim} and $P_{Diameter}$ are slightly better if many records are inserted into the QTree, we prefer $P_{MaxBound}$ because it performs best in the average case.

5.2.6 Evaluation

In this section we evaluate some more aspects of the QTree. As $P_{MaxBound}$ seems to be a good penalty function, we used $P_{MaxBound}$ in all the tests we discuss in the following. However, the general tendencies of our findings are still the same no matter what penalty function we use. At first, we examine the influence of the f_{max} and b_{max} constraints and the number of represented records on the approximation quality of a QTree. Afterwards, we examine the influence of the data distribution and the number of indexed dimensions. Furthermore, we discuss the influence of the order in which data records are inserted into the tree. And finally, we evaluate the deletion strategies we have proposed for the QTree.

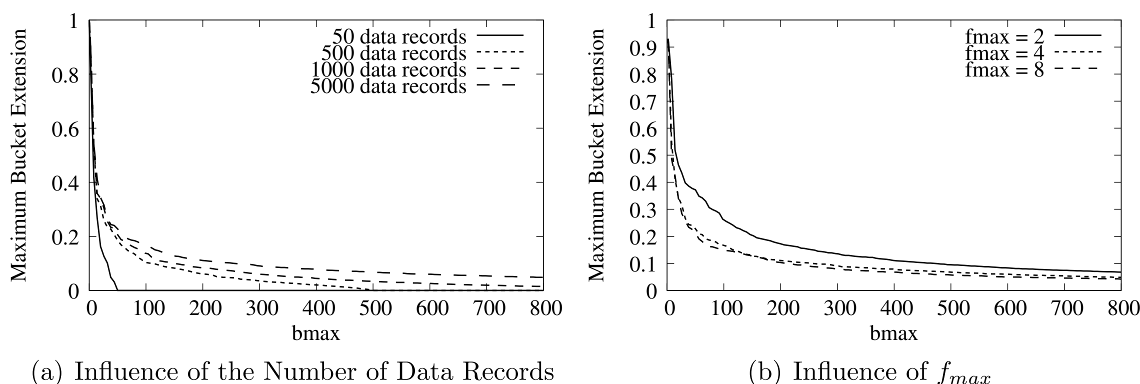
Influence of f_{max}, b_{max}, and the Number of Inserted Data Records

Intuitively, the approximation error should grow for higher numbers of inserted data records due to the bigger buckets that are necessary to represent all records. But how to measure such an approximation error? The key concept is to use the sizes of the buckets' MBBs as a measure. The greater the MBBs the greater is the approximation error of a record that is represented. Thus, we can measure the approximation error of a QTree by evaluating the maximum extension of any of its buckets. As an appropriate measure we use the *Maximum Bucket Extension* (*MBE*) in a QTree, which is computed as defined in Equation 5.11.

$$MBE(QTree) := \max_{B \in QTree} \left\{ \max_{\substack{d \in \{1,\dots,d_{max}\} \\ DimSpec[d].high \neq DimSpec[d].low}} \left\{ \frac{B.high - B.low}{DimSpec[d].high - DimSpec[d].low} \right\} \right\} \tag{5.11}$$

To illustrate the influence of the number of records that are inserted into a QTree, we did several tests with data sets of different sizes. For this purpose, we used two-dimensional random data sets (attribute values restricted to the interval $[0, 1000]$). We created three data sets consisting of 1000, 3000, and 5000 data records. We set f_{max} to 4, varied b_{max}, and measured MBE after inserting all records. The results that we obtained are illustrated in Figure 5.17(a). The general tendency is: the higher the number of data records, the higher is the approximation error. Although not shown here, we did the same tests for other data sets as well and observed that the effect we found for random data is negligible for other data sets such as clustered data if we only increase the number of records within a cluster but not the number of clusters themselves. The reason is that the approximation error is not influenced by the number of records a cluster consists of but by the number and size of the clusters – we will discuss this issue in more detail below.

The general tendency illustrated in Figure 5.17(a) is as logical as anticipated and holds for all data sets: the higher b_{max}, the lower is the approximation error. The reason is obvious and commonly known for all data summarizing structures: the more memory consumption (in case of the QTree determined by b_{max}) the lower the approximation error and thus the higher the approximation quality. If the number of buckets is equal or even higher than the number of records, the approximation error (MBE) is 0 because each record is represented by its own bucket.



(a) Influence of the Number of Data Records

(b) Influence of f_{max}

Figure 5.17: Influence of f_{max} and b_{max} on Random Data

We also examined the influence of f_{max} on the approximation error. The results for the random data set consisting of 5000 records and varying b_{max} are shown in Figure 5.17(b). The general tendency is that the approximation error is lower the more siblings a bucket may have. The reason is that we only consider sibling buckets for merging. Thus, the

higher f_{max} the better is the chance of finding a pair of sibling buckets whose mergence causes a minimal increase in the approximation error. Consequently, the approximation error tends to be lower for a higher fanout.

As indicated in Figure 5.17(b), another effect we found in our evaluation is that f_{max} should be greater than the number of indexed attributes of a data record, i.e., greater than the number of dimensions. Although we do not illustrate it in further diagrams, tests with other data sets support this finding. For the two-dimensional random data set the approximation error is clearly smaller for $f_{max} \geq 3$. The reason is that each node may have multiple children that "specialize" in different indexed dimensions.

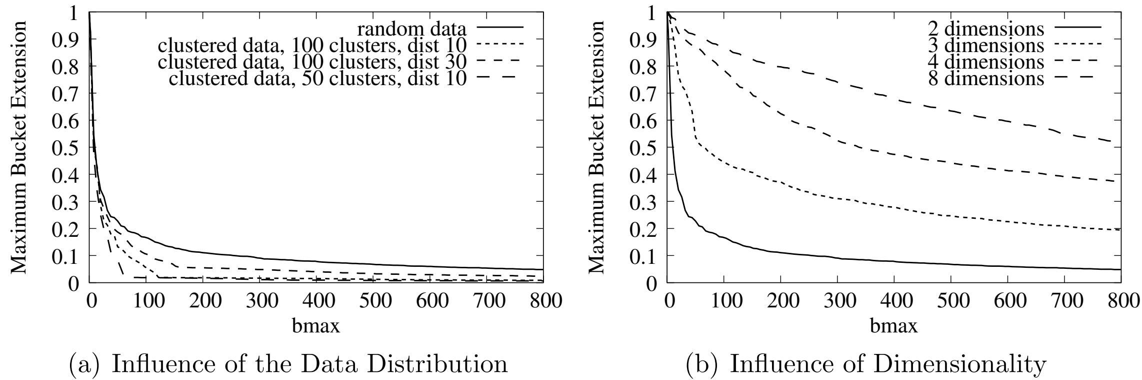
Data Distribution and Dimensionality

In further tests we examined the influence of the data distribution on the approximation quality. We used the random data set with 5000 records that we have already used above. Furthermore, we used several clustered data sets. Two of the clustered data sets consist of 100 clusters and 50 data records per cluster (5000 records per data set). Each record was created by adding a random offset of $[-dist, +dist]$ to the cluster center's coordinates – attribute values are again limited to the interval $[0, 1000]$. The difference between the two data sets is the *dist* value (10,30) that determines the extension (diameter) of the clusters. The third clustered data set that we consider in this section has the same number of records (5000) as the other two sets but fewer clusters: 50 clusters with 100 records each.

Figure 5.18(a) shows the results we obtained for these four data sets and varying b_{max} ($f_{max} = 4$). These results show that the characteristics of the data set have a strong influence on the approximation quality. Clusters with a small diameter can easily be represented by one bucket without resulting in a large MBB. The larger the cluster diameter the larger has to be the MBB of the representing bucket – compare the lines for the data sets with 100 clusters and different cluster diameters: *dist* 10 and *dist* 30. The higher the cluster diameter, the bigger must be the buckets that represent the clusters and thus the higher is the approximation error. Random data has the highest approximation error because there simply are no clusters that could be summarized with low approximation error.

Figure 5.18(a) also shows the influence of the number of clusters on the approximation error: the higher the number of clusters, the higher is the approximation error. The reason is that in case there are fewer clusters, there is less space that has to be described and represented by buckets. For example, if there are 100 clusters we need at least 100 buckets to represent each cluster with low approximation error. If we have only 50 clusters, we can represent each cluster with 2 buckets instead of only 1. Thus, the buckets are smaller and the approximation error is lower. In summary, as we have already pointed out above, the approximation quality does not depend on the number of records contained in the clusters but on their number and extensions.

We also examined the influence of the number of indexed dimensions on the approximation error. For these tests we used a random data set (5000 records) with 8 dimensions (restriction of attribute values again to the interval $[0, 1000]$) and indexed for each test a subset of them $(2, 3, \ldots, 8)$ – $f_{max} = 4$. Figure 5.18(b) presents the results. The general tendency is clear: the higher the number of indexed dimensions, the higher is the approx-



(a) Influence of the Data Distribution

(b) Influence of Dimensionality

Figure 5.18: Influences of Data Distribution and Dimensionality

imation error. The reason simply is that the data space grows with higher dimensionality and the buckets need to be larger in order to still summarize the data records.

Order of Insertion

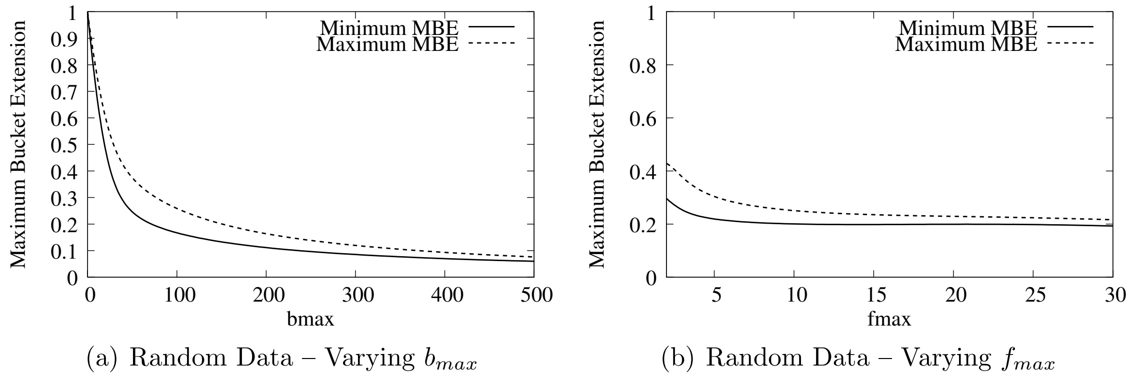
As a QTree is constructed incrementally by inserting one data record after another and the buckets grow to encompass the inserted data records, the locations and sizes of the buckets' MBBs depend on the order in which records are inserted. We also examined the influence of the insertion order with respect to varying b_{max} and f_{max}. For each setup we ran 1000 test runs and measured the maximum bucket extension (MBE) after the insertion of all 5000 records of the two-dimensional random test data set. We used $b_{max} = 150$ and $f_{max} = 4$ as parametric setup. Figure 5.19 shows two lines, one representing the worst MBE values we obtained in the 1000 test runs and the other one representing the best MBE values. By picturing both in the same diagram, we obtain a visualization of the fluctuation in the approximation error caused by the order of insertion.

Figure 5.19(a) shows our results with respect to varying b_{max} and Figure 5.19(b) the results with respect to varying f_{max}. In these results we again see the tendency that we have already found above: the higher b_{max}, the better is the approximation quality and if f_{max} is higher than the dimensionality, we obtain better approximations. Furthermore, we see that the approximation error up to a certain extent depends on the order of insertion. The variation is smaller for high numbers of buckets because there are more buckets that cover the data space and the influence of each single data record is smaller. As further tests have shown, these findings do not only hold for random data but also for other data sets such as clustered data.

Although the dependency of the approximation quality on the order of insertion can be considered a weakness, remember that bucket boundaries are variable and chosen in dependence on the data that is represented.

Influence of Deletion

Finally, let us discuss the performance of the update strategies we propose for deletion (Section 5.2.4). For these tests we used the random data set and the clustered data set (100 clusters, $dist = 10$) with 5000 records that we have already used in previous



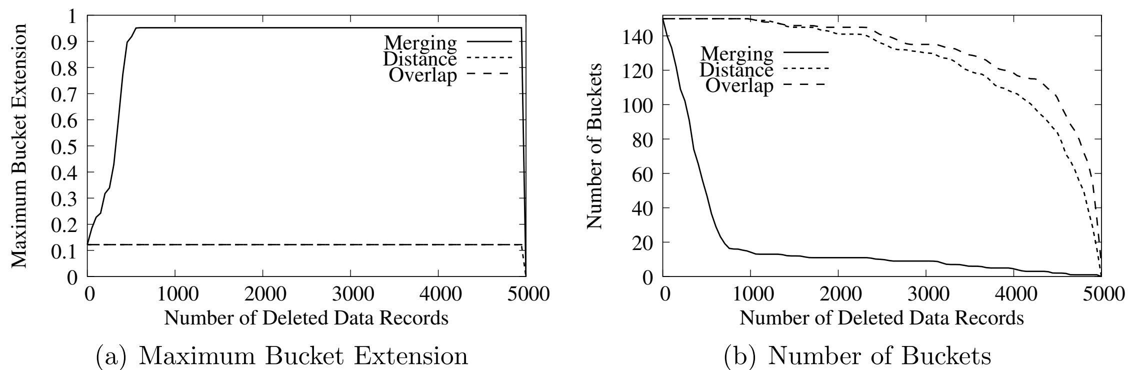
(a) Random Data – Varying b_{max}

(b) Random Data – Varying f_{max}

Figure 5.19: Influence of the Order of Insertion on the Approximation Error

tests. The QTree parameters were set to $f_{max} = 4$ and $b_{max} = 150$. In our first tests we created a QTree and then deleted one record after another. We evaluated all three deletion strategies we have discussed in Section 5.2.4. The results we obtained for the two data sets are presented in Figures 5.20 and 5.21.

Let us first consider the results for the random data set (Figure 5.20(a)). The maximum bucket extension (Equation 5.11) grows only if we apply the “merging” strategy because this is the only strategy that merges buckets upon deletion. The other two strategies do not affect the buckets' MBBs but only their statistical entries. Figure 5.20(b) shows how the number of buckets used by the QTree changes when deleting records. For the merging-based strategy the number of buckets is reduced much faster than for the other two strategies. The reason is again that buckets are merged in order to execute a deletion. Applying the other two strategies, buckets are only deleted when their statistical entries are reduced to 0. Thus, it takes much longer for these strategies to reduce the number of buckets. In comparison of these two, the distance-based deletion strategy reduces the number of buckets faster than the overlap-based strategy.



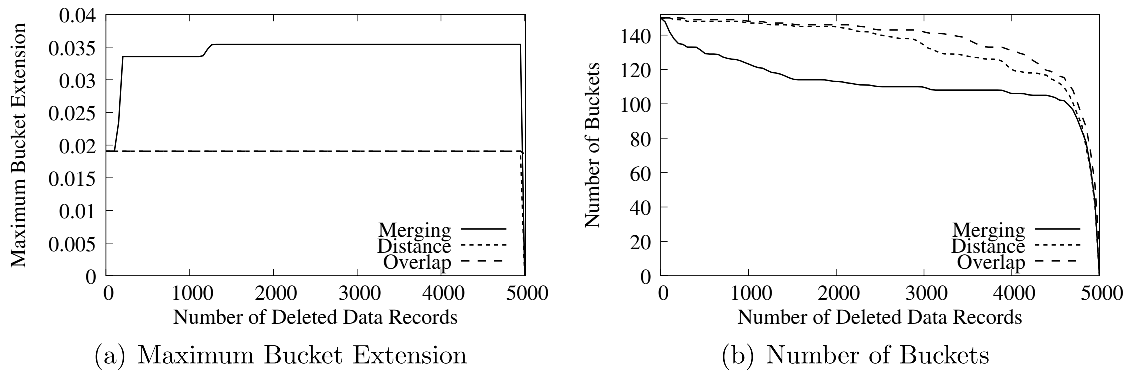
(a) Maximum Bucket Extension

(b) Number of Buckets

Figure 5.20: Influence of the Deletion Strategy – Deleting Single Records from the Random Data Set

Figure 5.21 shows the results we obtained for the same test runs on the clustered data set. Again, the maximum bucket extension only changes for the merging-based strategy and remains the same for the other strategies. Note that the y-axes of Figures 5.20(a)

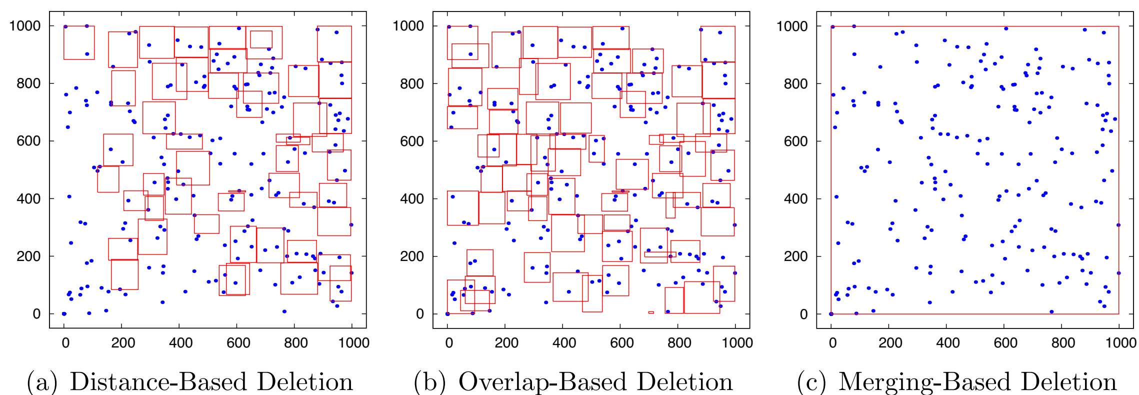
and 5.21(a) have different scales and the growth in the maximum bucket extension for clustered data is still low. Figure 5.21(b) illustrates the reduction in the number of buckets. Again, the distance-based strategy shows a faster reduction in the number of buckets than the overlap-based strategy and the merging-based strategy the highest reduction.



(a) Maximum Bucket Extension (b) Number of Buckets

Figure 5.21: Influence of the Deletion Strategy – Deleting Single Records from the Clustered Data Set

So far we have only considered the influences of deletion on the maximum bucket extension and the number of buckets. As an increase in the maximum bucket extension is in general disadvantageous for the performance of the query processing strategy, one might tend to prefer the distance-based strategy. However, the reduction in the number of buckets should also be taken into account. But the most important issue has so far been left out: are all records still represented correctly after executing deletions? To illustrate the answer to this question, Figure 5.22 depicts the QTrees after the deletion of 4800 records.



(a) Distance-Based Deletion (b) Overlap-Based Deletion (c) Merging-Based Deletion

Figure 5.22: QTree after Deletion of 4800 of 5000 Records – Random Data

After having deleted so many records from the tree, the side-effects of the deletion strategies are most obvious. Figures 5.22(a) and 5.22(b) reveal that both, the distance-based and the overlap-based strategy, sometimes delete records from the "wrong" buckets. As a consequence, both figures show that some records (that have not yet been removed

from the tree) are not represented by any bucket. This is the worst case for any query processing strategy that relies on the correctness of the information provided by the routing indexes. Were these QTrees part of a routing index, we would retrieve incomplete results if the corresponding neighbor was wrongly pruned from consideration due to false negative routing decisions. With respect to this notion of correctness, the merging-based strategy (Figure 5.22(c)) performs best. Applying this strategy, it is impossible that records are not correctly represented by any bucket. We have conducted several test runs with other data distributions and larger regions that are deleted instead of single records. For all these tests we found the same tendencies.

Since we want to use the QTree to build routing indexes, we have to insist on absolute correctness and discard the overlap-based and distance-based strategies. However, we are aware that especially in the case of random data this could lead to difficulties that we have to compensate, e.g., by increasing b_{max} and thus reducing the degree of overlap between buckets. But nevertheless, we have a routing index whose correctness we can rely on.

5.2.7 Extension to Support String Attributes

To complete our discussion on base structures for routing indexes and the QTree, let us briefly discuss how to index string attributes. Whereas for numerical data it is straightforward how to define similarity between attribute values, this is not the case for string attributes. For the latter a widely used similarity function is the edit distance, or the Levensthein distance [121] respectively, so that the distance between two strings is defined by the minimum number of edit operations (update, deletion, insertion) that are necessary to transform one string into the other.

In order to capture string data, the literature proposes several approaches, e.g., tries [45,65] and patricia tries [139]. Both approaches are prefix tree variants, the characters of a string are represented by inner nodes and the string itself is stored in a leaf node. A specific string can be found by traversing the tree in dependence on the sequence of its characters. There are also some approaches that consider approximate string matching based on the edit distance for tries, e.g., [175] using dynamic programming. Unfortunately, prefix trees and their variants do not summarize/aggregate data, which means they cannot be used as a base structure for routing indexes. However, we could still introduce aggregation by replacing subtrees with statistical nodes – similar to the QTree in comparison to the R-Tree. If we did so, we could no longer determine the edit distance for the represented strings but only for their prefixes represented by nodes that have not been replaced.

Another approach to represent strings are (positional) q-grams [181, 192, 195]. The principle is to determine all possible q-elemental substrings of a string, e.g., the set of q-grams of length 3 ($q = 3$) for the string "house" would be: $__h$, $_ho$, hou, ous, use, $se_$, $e__$ or in case of positional q-grams $(1, __h)$, $(2, _ho)$, $(3, hou)$, $(4, ous)$, $(5, use)$, $(6, se_)$, $(7, e__)$ with the number denoting the position of the q-gram in the original string. The similarity with respect to the edit distance between two strings can be determined based on their sets of q-grams [181, 182]. The intuition is that strings with small edit distances share a large number of q-grams. A detailed comparison of positional and non-positional q-grams for selection and join queries in RDBMSs is given by Gravano et al. [75, 76]. It is

possible to use q-grams to build routing indexes by merging the q-gram sets of multiple strings. The resulting superset can be used in conjunction with the edit distance to determine if it is possible that a specific string (or one with a specific edit distance similarity) was among the original strings whose q-gram sets have been merged. Due to the merging, similar to indexes for numerical data, this kind of routing index yields the possibility of false positive routing decisions but not false negatives.

For our application scenario it would be worthwhile not only to index string attributes and numerical attributes in isolation but also to capture the correlations between them. In order to use histograms or QTrees for this purpose, a straightforward solution is to transform strings into numerical space. One basic approach would be to use the ASCII code of the characters and use one dimension for each position in the string. This means we would have one dimension for each numerical attribute and multiple dimensions (their number determined by the maximum number of characters of a string) for each string attribute. As using this technique for string attributes would require a high number of dimensions, it would be hard to find any clusters (curse of dimensionality [16]) and structures such as the QTree and histograms would not be efficient.

An alternative would be the application of hash functions on the strings to obtain numerical values. The problem with hash functions is that although there are some that are neighborhood preserving with respect to the lexicographic order of strings, we do not know of any hash function that preserves neighborhood with respect to the edit distance. Thus, the edit distance would not be applicable as distance function.

The solution we propose [113] to index numerical and string attributes within the same structure – capturing correlations between them – is based on q-grams. Additional information about string attributes is added to a bucket, i.e., for each indexed string attribute a a bucket b holds a set of q-grams that represents the union of all q-gram sets generated with respect to a for the records represented by b. As it is likely to be inefficient to store the q-grams themselves within a bucket, we propose to use a bitvector representation. In the simplest case, a specific bit exclusively represents a specific q-gram. Alternatively, hash functions can be used to reduce the size of the bitvectors – the consequence is an increase in the number of false positive routing decisions as one bit might represent multiple q-grams and there is no indication which one of them actually caused the bit to be set to 1.

This solution can be optimized in several ways. First, as the buckets of the QTree are constructed in consideration of only the numerical attributes, the string attribute values of the inserted records can be very different. Consequently, there are many different q-grams and thus there is a higher risk of making false positive routing decisions. Furthermore, whereas it is straightforward how to adapt the bitvectors for the insertion of records, the deletion is impossible as we do not know if a specific q-gram was exclusively contained in the q-gram set of the deleted record or also by other q-gram sets of records that are still represented – reconstruction or storing additionally the number of records whose q-gram sets contain a specific q-gram are possible but expensive solutions to the problem.

In the context of this dissertation, we do not go into details on how to index string data and the combination of numerical and string attributes. Our intention was to sketch a possible solution to the problem. Improving this basic approach or finding even better solutions to the problem is part of our future work and not considered in the following.

5.2.8 Summary

In summary, in this section we found out that the QTree does efficiently what we developed it for: representing data with "small" buckets. Subspaces that do not contain any data records are left out such that the query processing strategy can efficiently make use of the DDSs's information (Chapter 6). We found out that the merging-based deletion strategy is most suitable for our purposes.

We have already mentioned and discussed several improvements of the QTree such as the extension to string attributes. Moreover, we have reviewed several variants of penalty functions that can be used to determine which pair of buckets causes the least penalty when merged. We found out that $P_{MaxBound}$ is the best solution for our requirements. However, so far we have limited our considerations to only sibling buckets. In future work we might extend this concept to merging non-sibling buckets. This should minimize the influence of f_{max} on the approximation quality and allow us to find optimal pairs of buckets for merging. Another issue that might be worth being considered in future work is to allow QTree nodes (their MBBs resp.) to assume other more intricate shapes – so far we focus on rectangular bounding boxes, which makes it low-effort to determine whether two MBBs overlap.

5.3 Maintenance

After having introduced the notion of Distributed Data Summaries (DDSs) and an example based on the QTree in the previous sections, let us now focus on the often neglected problem of keeping routing indexes up-to-date. The reason why we need to face such problems is that due to the inherent dynamic behavior of P2P-based systems the network structure as well as the peers' local data might change over time. Consequently, the data described by the routing indexes, i.e., the data accessible via neighboring peers, is likely to change after having created the initial set of summaries. Overreliance on their correctness without taking any measures to keep them up-to-date sooner or later results in false routing decisions. For example, if for a given query a peer decides – based on the information provided by its routing index – not to forward the query to a specific neighbor, data that is relevant to the query might be missed if the neighbor acquired relevant data in the meantime (false negative routing decision). Hence, the result set for a query might be incomplete. On the other hand, if a peer queries a neighbor that no longer provides relevant data, query execution costs are unnecessarily increased (false positive routing decisions). Therefore, when peers change their local data or the set of neighbors, it is necessary to have all neighbors (that hold summaries describing their data) update their routing indexes. The question is how to do that efficiently.

To solve this problem, there are basically two straightforward extremes that are both unacceptable: first, not updating routing indexes at all but minimizing network load and second, creating a new set of routing indexes for all peers whenever an update occurs [41, 53]. Hence, we identify two contrary goals: (i) keeping routing indexes up-to-date and (ii) minimizing network load. In the remainder of this chapter, we try to find an acceptable tradeoff between these two goals. However, as peers might have different schemas, we begin with a description of how updates can be encoded and how they can be rewritten from one schema into another. We then proceed with a discussion

on how to efficiently construct routing indexes. Furthermore, we propose and classify several strategies that can be used to solve the problem of updating them for numerical data. Although we focus on DDSs based on QTrees and equi-width multidimensional histograms, most of the techniques we present can be extended and applied to the general case. In the following we use the terms DDS and routing index interchangeably.

5.3.1 Update Representation and Rewriting Updates as well as Summaries

There are only three types of updates to data records that need to be considered: deletions, insertions, and changes of attribute values – each is defined by a set of indexed attribute values and a keyword indicating the update type (if the order of attributes is fixed, we do not need to include the attribute names). To encode such updates, we only need deletions and insertions. Changes of attribute values are encoded as a deletion followed by an insertion. A list of such deletions and insertions is referred to as *Add-Remove-List (ARList)*.

Updates can be merged into a more compact representation defined by a region r and a correction value indicating the number of updated records within. However, we should not use arbitrary regions for this purpose because such a region would most likely overlap multiple buckets of the neighbor's routing index – splitting up the correction value among these overlapping buckets would only lead to approximation errors. Thus, we determine the regions in dependence on the buckets of the routing index so that each updated region r is completely contained in one of the neighbor's buckets. For this purpose, each peer maintains a summary of its local data that has the same structure as the base structure of the neighbors' routing indexes. This local index can be obtained as a by-product from the construction phase (Sections 2.3 and 5.3.2), in which these local summaries are propagated through the network as a description of the data provided by the peers.[2] By adapting that local summary whenever local updates occur, we need to distinguish four cases:

1. a new bucket has been created,
2. an existing bucket has been deleted,
3. an existing bucket has been changed, and
4. a bucket has not been changed.

In the context of equi-width multidimensional histograms a new bucket is said to be created if it has been empty before. Likewise, a bucket is said to be deleted if it is empty in the new version but not in the original version. For each bucket that corresponds to any of the first three cases we create one update entry that describes the bucket's boundaries and difference in the number of represented elements (positive value for case 1, negative value for case 2, and positive or negative value for case 3).

Updates are merged only if the merged version reduces the volume of the corresponding update message. Otherwise, an ARList is used. Furthermore, note that by merging

[2]As the regions of this index have been used to construct the neighbors' routing indexes, the update region is always completely contained in one of the routing indexes' buckets. Thus, in the case of the QTree and deletion applying the merge strategy (Section 5.2.4), we only need to merge buckets that completely enclose the updated region instead of all buckets that at least partially overlap. In case of equi-width multidimensional histograms, the bucket boundaries are the same for all peers, anyway.

updates it is possible that two updates compensate each other. Whenever this is the case, we can safely prune the set of compensating updates. The same technique can be used to merge updates originating from neighboring peers. In the case of establishing new links or removing old ones, all buckets in the local routing index that describe affected data are sent as updates to all the other neighbors.

Still, there is another problem we have to deal with in PDMSs: as peers are allowed to use different local schemas, not only queries need to be rewritten but also updates that are propagated from one peer to another. In order to decide efficiently which neighbors provide relevant data to a query (Chapter 6) (based on the routing indexes and the evaluation result of the query on the local data), we assume a routing index to be defined on the local schema of the peer that holds it. This means data summaries as well as data updates need to be rewritten from the local schema into the schema of the receiving peer.

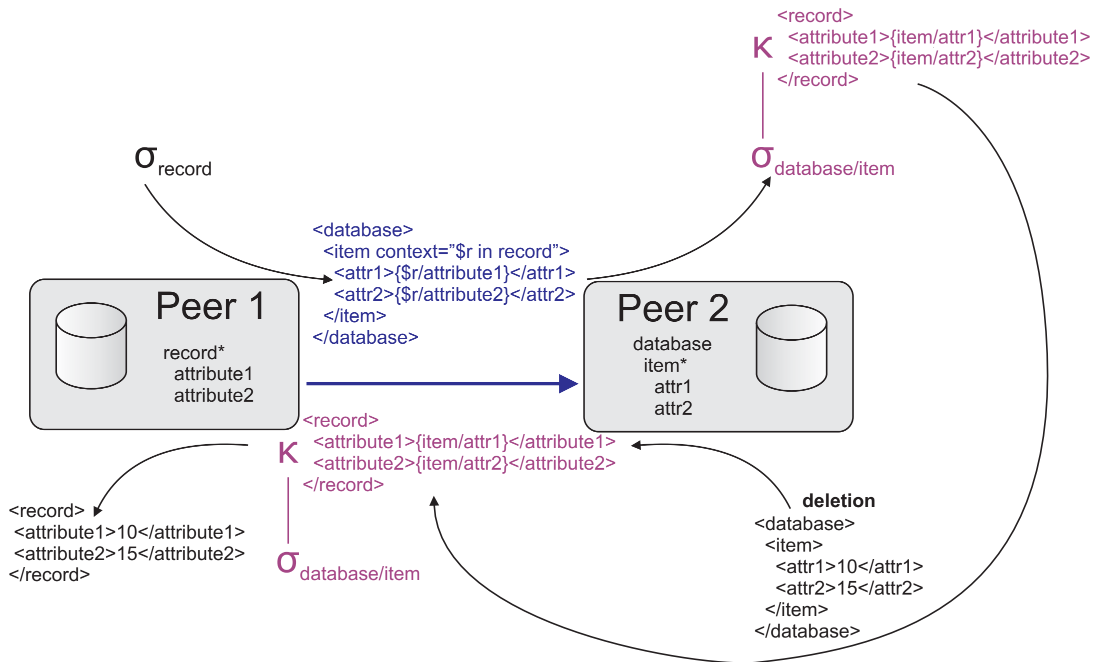


Figure 5.23: Rewriting Updates in the Presence of LAV Mappings

Assume we face the abstract example situation sketched in Figure 5.23 and P_2 deletes one of its local data records. In this example there is only a mapping from P_1 to P_2. This means P_1's routing index describes the data stored at P_2 but not the other way around. Consequently, P_2 needs to send updates to P_1 even though it does not hold an appropriate mapping. To solve this problem in the presence of LAV-style mappings, we introduce the concept of *permanent update request queries* (*PURQs*). They should be installed already when mappings between two peers are defined. Assume P_1 has issued such a query and has sent it to P_2. A PURQ is a simple selection query that selects all elements that P_1's routing index is defined on. In contrast to standard queries, it is issued only once but evaluated at the receiver every time it updates its local data.

A PURQ is rewritten using the mapping (Figure 5.23). The rewritten PURQ describes how to transform updates from the schema of P_2 into the schema of P_1 and is consequently

used for this purpose any time P_1 updates its local data. Note that an update received from another neighbor of P_2 can be rewritten using the same technique because the records contained in the update message use P_2's local schema. In contrast to standard query processing, the rewritten PURQ is only evaluated on the portion of the data that has been updated instead of evaluating it on all the local data of P_2. If an update at P_2 does not affect any of the attributes the rewritten PURQ refers to, it does not have to be evaluated at all because there cannot be any changes to the data summarized by P_1's routing index.

In order to construct routing indexes, peers only need to forward the buckets describing the data, i.e., in the case of QTrees there is no need to send inner nodes. Each region/bucket is defined by a set of attributes and a set of intervals (each corresponding to one of the attributes). We can rewrite these attributes using the techniques discussed above. Since our mappings do not contain any functions or data transformations, we do not need to adapt the buckets' regions.

5.3.2 Constructing Distributed Data Summaries from Scratch

A problem we have to deal with in unstructured P2P networks is that arbitrary networks might contain cycles. If we index the data nevertheless, the data of a peer P_0 might be contained multiple times in the routing index of a peer P_3, for example once for its neighbor P_1 and once for its neighbor P_2 – if P_0 is reachable via both neighbors. With respect to the strategies for distributed query processing, which we propose in Chapter 6, this is not a problem for skyline queries as the decisive characteristic for them is the dominance relation (Section 2.1.2, Definition 2.1.3) and all relevant neighbors are queried. Thus, the worst thing that could happen is that the result set contains duplicates (false positive routing decisions).

In case of selection queries or range queries P_3 would possibly also make false positive routing decisions, e.g., forwarding the query to both P_1 and P_2 although it only needs to retrieve the data from P_0 once. However, as P_3 relies on its routing indexes, it will not be able to identify duplicates before it receives the results from both neighbors. Thus, again despite false positive routing decisions the correct and complete answer to the query would be retrieved.

For top-N queries we need to estimate the number of records that each neighbor provides. Based on this information, P_3 determines the minimum number of neighbors that need to be queried in order to answer the query. The problem is that when considering the same data twice, it is possible that a neighbor P_4 is not queried although after having removed all duplicates in the result set, it is clear that the data of P_4 should have been considered (false negative routing decision). In that case P_3 needs to rerun query optimization and forward the query to additional peers such as P_4. With respect to the general steps of query processing (Figure 5.1) this means that P_3 needs to go back from step 6 (postprocessing) to step 4 (optimization).

As we will see later in Chapter 6, the same considerations hold for the application of relaxation, i.e., cycles lead to false positive routing decisions for skyline queries and to false negative and false positive routing decisions for top-N queries. Although the techniques for cycle detection that we have discussed in the context of query rewriting in Section 4.3 might help identify false positive routing decisions and reduce the number

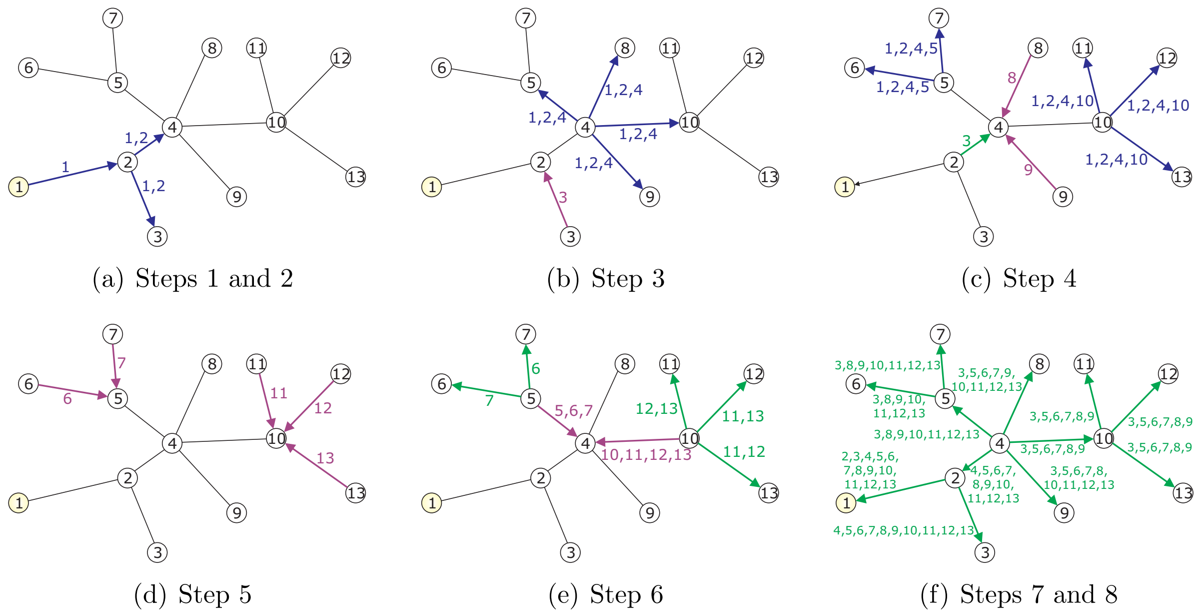
of duplicates, they cannot entirely solve the problem. Thus, although in principle our algorithms can deal with cycles, we try to avoid them and index the data in a cycle-free network.

In a network that has not yet established DDSs or routing indexes, in general the only way of creating them is to flood the network in a strategic way. However, gossiping [42,47] could be an alternative, which we do not consider in this context because it might take a long time until routing index construction is completed and peers highly rely on the freshness of their routing indexes in order to process queries efficiently (Chapter 6). A basic strategy to construct routing indexes is to have one peer decide when it is time to (re)create all summaries in the network. At first, let us consider an acyclic network where we have bidirectional mappings for each link between peers. The principle is to flood the network and have the peers exchange information about the data they store locally. For this purpose, we have to distinguish between three message types: *Update Request Messages*, *Update Answer Messages*, and *Update Compensation Messages*. The goal is to send information through the network that describes what data can be retrieved by which neighbor.

Figure 5.24 illustrates the principle with an example. Assume peer P_1 initiates the process. Then, it computes a summary of its local data and sends it to all its neighbors. Such a neighbor, in this case only P_2, in turn computes a summary of the received information with its local index (aggregating both summaries into just one) and propagates the result with an update request message to all its neighbors (flooding) – Figures 5.24(a) and 5.24(b). P_2 can already construct a part of its routing index by using the summary received from P_1 to describe the data it can retrieve by forwarding a message to P_1. All summaries that are sent through the network adhere to the same base structure (e.g., an equi-width histogram) with the same parametric configuration (e.g., the number of buckets per dimension). This guarantees that the summaries contained in different messages can be combined and do not provide insufficiently detailed information.

Each peer that receives an update request message and has no other neighbors, e.g., P_3 in Figure 5.24(b), sends an update answer message containing a summary of its local data in response. These answer messages are also propagated through the network – albeit in a slightly different fashion: a peer starts propagation only if all its neighbors except one have already sent answer messages. In Step 5 (Figure 5.24(d)) P_{10} receives answer messages from all its neighbors except one and consequently sends an own answer message to P_4. This message contains a combined summary of its local data and all the data provided by its neighbors, i.e., all the data that can be accessed by P_4 via P_{10}. Consequently, the summary represents part of P_4's routing index. At the same time P_{10} sends update compensation messages (Figure 5.24(e)) to all its other neighbors. These messages contain information about the data that can be accessed via P_{10} and has not yet been propagated with the update request message. After all peers have received such update compensation messages, the (re)creation process ends and all the peers in the network have up-to-date distributed data summaries.

Considering arbitrary networks we have to deal with cycles and consequently with the problem of termination. To solve this problem, we could use a time-to-live (TTL) value that determines the maximum number of hops that are indexed (horizon) for each peer. This means that for the construction process we can no longer aggregate all summaries that describe the data accessible via a specific peer before propagating it to its neighbors.



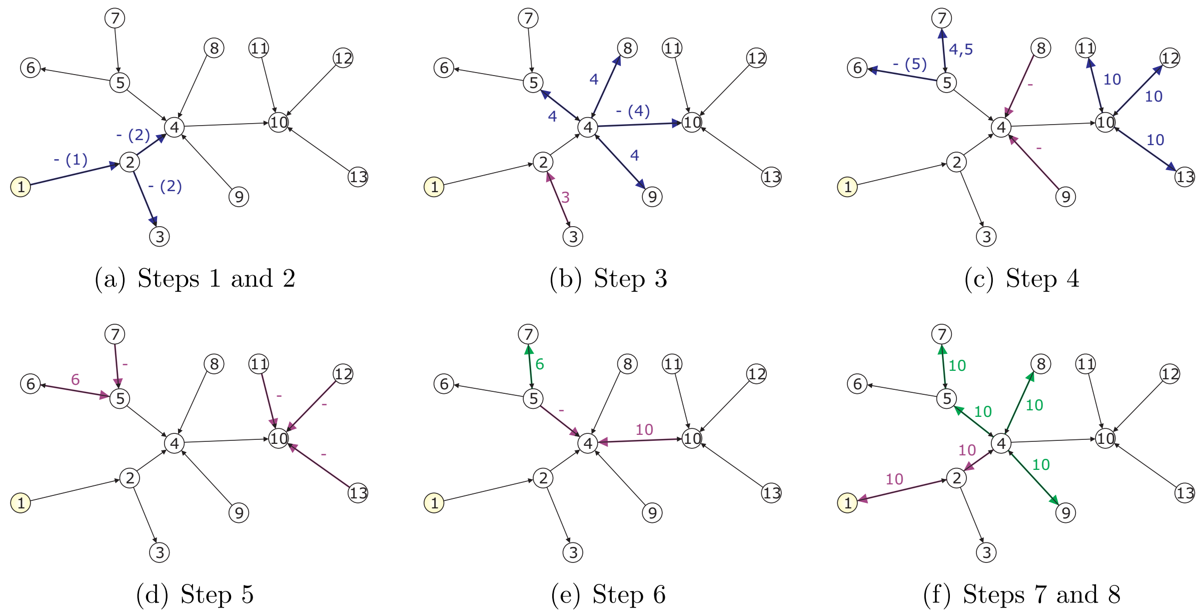
(a) Steps 1 and 2 (b) Step 3 (c) Step 4

(d) Step 5 (e) Step 6 (f) Steps 7 and 8

Figure 5.24: DDS Construction with P_1 as Initiator. *3 Message Types: Update Request Messages (blue), Update Answer Messages (red), Update Compensation Messages (green)*

In contrast, at each peer we need to prune some of the summaries because the distance of the peers they correspond to could exceed the TTL value of the neighbor. Consequently, the construction process would be more expensive because update propagation messages would have to contain multiple summaries (instead of only one aggregated summary) so that according to the horizon summaries can be pruned. A second solution to solve the problem could be the application of a spanning tree algorithm that eliminates cycles or rather the integration of a spanning tree algorithm into the construction process. In that case, when a peer receives an update request message from two of its neighbors, it could simply answer the second request with a denial message so that its data is only indexed for one of the two routes. As it took longer for the second request to arrive at the peer, "deactivating" the link does not impair query processing because the other link should be favored for query processing anyway. However, in consideration of information loss caused by different mapping paths, more sophisticated criteria could be considered to choose the best mapping path. In any case, if the favored link fails, the peer could still "remember" the deactivated link and the existence of the cycle. Because of the simplicity and low execution costs, we favor the spanning tree-based variant over the horizon-based variant.

As mentioned above, we assume each peer to have installed a permanent update request message (PURQ) to all peers that it holds mappings to. Hence, in the case of bidirectional mappings we can simply use the rewritten PURQs to rewrite the summaries and run the algorithm as described above. What we still need to discuss is whether this also works for unidirectional mappings. Under the assumption that we have unidirectional mappings but still an adjacent network, the main difference is that update request messages do not necessarily contain summaries, they only do so if the receiver holds a mapping to the local schema of the sender. Figure 5.25 shows an example.



(a) Steps 1 and 2
(b) Step 3
(c) Step 4
(d) Step 5
(e) Step 6
(f) Steps 7 and 8

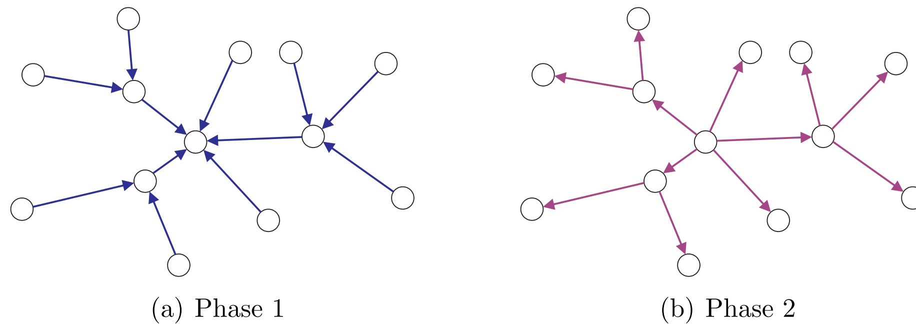
Figure 5.25: DDS Construction in the Presence of Unidirectional Mappings with Peer P_1 as Initiator. 3 *Message Types: Update Request Messages (blue), Update Answer Messages (red), Update Compensation Messages (green)*

In this example, again P_1 initiates the construction process. If it is aware that P_2 has no mapping to P_1, the update request message does not contain P_1's local summary. If P_1 is not aware, it sends the summary along with the message. In any case P_2 is aware that it has no mapping to P_1, hence it does not forward P_1's summary along with its update request message to P_3 and P_4. The second difference for the unidirectional case is that update answer messages, too, do not necessarily contain summaries, e.g., it would not make sense for P_8 to send its local summary to P_4 (Figure 5.25(c)) as P_4 does not have a mapping to P_8 – the received update request message has already contained this information. If all peers are aware whether their neighbors hold a mapping to their local schema, all update answer messages that do not contain a summary can be saved. After having processed the last update compensation messages, each peer holds a routing index that correctly describes the data accessible via the neighbors it holds mappings to.

We are aware that this strategy can be improved in several ways. One of them is called *2-Phase-Flooding*. It is based on the assumption that multiple peers begin with the construction process at the same time – assuming that the network is cycle-free. This is for example possible when the index (re)creation takes place on a regular basis. In the following we focus on bidirectional mappings as the adaptation to unidirectional mappings is straightforward and similar to the ones we have already discussed for the baseline algorithm above.

Applying 2-phase-flooding each peer that has only one neighbor begins with sending its local data summary to its neighbor (using an update request message). Each peer that receives such a message uses the summary to construct its local routing index. Once a peer has received such messages from all its neighbors except one, it computes a summary of its local data and the summaries received along with the messages. The peer then forwards

the result to the one neighbor that has not yet sent a request message. Figure 5.26(a) shows an example. When a peer has received request messages from all its neighbors, the second phase, in which update compensation messages are sent (Figure 5.26(b)), begins. These messages again contain information that could not be forwarded in the first phase. Once all peers in the network have received such compensation messages, the construction process ends and each peer has a set of up-to-date summaries of the data that can be accessed by neighboring peers, i.e., each peer has an up-to-date routing index.



(a) Phase 1 (b) Phase 2

Figure 5.26: DDS Construction – 2-Phase-Flooding

5.3.3 Classification of Maintenance Strategies

After having created the initial set of routing indexes, peers might update their data. Consequently, routing indexes need to be updated. We assume updates to be encoded as described in Section 5.3.1. However, note that this is only possible for routing indexes whose base structures support incremental maintenance, i.e., it is possible to encode an update and adapt the summary accordingly. The alternative, as proposed by Crespo et al. [41] (Section 2.3), is a partial reconstruction of the routing indexes, i.e., update messages do not only contain an encoded update but a complete summary describing the data that can be accessed via the sender. This means that both the computational effort as well as the size of an update propagation message is clearly higher in comparison to the alternative.

However, in the following we assume that the base structure of the routing index supports incremental maintenance. If so, we still have to solve the problem of multiple possibly simultaneous updates. If each peer in a network of n peers with bidirectional mappings updates its local data only once, then by following a simple flooding approach n^2 messages have to be processed in the network, i.e., n per peer. If peers update their local data even more often, we have to assume considerable temporal delays because messages need to wait in the waiting queue of a peer before they can be processed. Remember that in parallel to handling updates, queries have to be processed. Thus, when updates do not occur only rarely, the network is primarily occupied with processing update messages instead of processing queries.

Update propagation in P2P systems can be regarded as a replication problem where some peers have replicas of other peers' data. In this case replicas are not exact replicas but only summaries of the original data. A lot of techniques have been proposed

for traditional replication problems in distributed environments [172]. Although the understanding of a replica is not the same, the update strategy is a similar problem that has to be solved in absence of any kind of global knowledge. Traditional synchronous solutions to manage distributed data usually handle only a small number of peers and are not suitable for widely distributed environments such as P2P systems, where network connections are unreliable and peers are temporarily unavailable [77]. Epidemic algorithms [57] seem to be a good solution for distributed environments since they work asynchronously, are robust against peer failures, do not make any assumptions on the network structure, and do not need any central instance for coordination. Using epidemic algorithms, each peer propagates the updates to a random subset of its neighbors. In contrast, in a PDMS peers should receive update messages with a minimal delay so that the routing indexes of all peers are as up-to-date as possible. Thus, for PDMSs, we prefer a more explicit strategy that enables to control or to guarantee when and to what degree routing indexes are updated.

However, neither propagating each single update by flooding nor not propagating updates at all can be an appropriate and efficient solution. In contrast, the freshness of routing indexes is vital for distributed query routing and thus for query processing (Chapter 6) so that we need to find a tradeoff between the two extremes. For this purpose, we identify three classes of maintenance strategies:

Update-Driven Propagation: A straightforward solution to the problem is to propagate each update through the network as soon as it is performed (*immediate propagation strategy*). An alternative is to accumulate a certain number of updates before propagating them to neighboring peers (*threshold propagation strategy*) or to compare the current data distribution of a sample to the distribution represented by the routing index in order to determine a discrepancy that requires an update [4]. Thus, the distinguishing characteristic of strategies in this class is that peers actively propagate updates through the network when a strategy-specific criterion is fulfilled.

Query-Driven Propagation: An alternative to propagating updates triggered by (accumulated) updates is to propagate them triggered by queries. A corresponding strategy can for example estimate the result cardinality based on the information provided by the DDSs. If the number of expected records is forwarded along with the query to a neighboring peer, the receiver can compare it to its answer and decide whether it is necessary to have the sender's routing index updated or not – the updates can directly be attached to the answer message (*query estimation strategy*). Alternatively, the sender of the query could directly use the query results received from its neighbors to update its routing index by applying query feedback (*query feedback strategy*).

Update Propagation on Demand: For propagation strategies of this class, a peer actively demands updates from its neighbors by sending demand messages. This might be triggered for example by a time interval so that updates are demanded in predefined time intervals.

In the following, we provide detailed information and examples of strategies for the first two classes, i.e., update-driven propagation and query-driven propagation. We do

not elaborate on the third one because it is straightforward and relatively inflexible. Furthermore, it requires sending additional messages (demand and update) and a peer would likely send demand messages more often than necessary as it has no indication whether neighbors hold updates or not.

5.3.4 Update-Driven Strategies

As already mentioned above, when applying update-driven maintenance strategies update propagation is triggered by updates. These are either updates of a peer's local data or updates received from neighboring peers. The main difference between the strategies in this class is *when* to propagate these updates. In the following, we present two strategies: a naïve strategy that propagates updates as soon as they are known and an advanced strategy that uses thresholds to reduce the number of update messages. Finally, we discuss some measures that peers adhering to the latter strategy might use to decide whether to propagate updates or not.

Immediate Propagation Strategy (IPS)

Propagating updates immediately is the most primitive strategy. When a peer updates its local data, the update is encoded and instantaneously forwarded to all neighbors. The same procedure is applied to incoming update messages from neighbors: routing indexes are updated and the updates are immediately forwarded to all other neighboring peers. The only difference is that now the peer needs to take care not to send the updates to the neighbor that they have been received from in the first place. Algorithm 12 summarizes this basic algorithm for IPS.

Algorithm 12 *processUpdatesIPS(Index index, updates, neighbors)*

```
1: index.insert(updates);
2: for neighbor ∈ neighbors do
3:    if neighbor.getPeerID() != updates.getSenderID() then
4:       neighbor.sendUpdateMessage(updates);
5:    end if
6: end for
```

Even without any experimental evaluation it is clear that this strategy is not efficient in a network with high update rates. But in networks with low update rates and the requirement that indexes should be as up-to-date as possible, this strategy might nevertheless be an alternative. Another advantage, apart from being up-to-date and simplicity, is that we only require a minimum of local memory since in contrast to the threshold strategies, which we discuss below, this strategy does not need to remember any past updates or compare the current index to any previous versions. Furthermore, applying this strategy a peer does not need and consequently does not hold an index for its local data. As a consequence, local updates are not merged (Section 5.3.1).

Threshold Propagation Strategy (TPS)

In contrast to IPS, threshold-based strategies forward updates not until a certain amount of them has been accumulated. To decide whether to forward updates or not, a peer uses

a set of propagation rules that determine when enough updates have been accumulated. Such a rule may for example be defined on the special characteristics of the base structure of the routing index, e.g., in case of a QTree on the extensions of the buckets that are affected by the updates. Assume we have two neighboring peers P_1 and P_2. Then, the summaries that are part of P_1's routing index have been used in the construction process to build the summary of P_2's routing index that describes the data accessible via P_1. Thus, it is a good idea to define propagation rules with respect to the change in the local routing index. However, a peer not only needs to consider the change in its routing index but also the changes made to its local data. To enable the use of the same propagation rules to local data updates as well as on updates to a peer's routing index, each peer maintains a local summary of its local data. This summary uses the same base structure as the routing index and can be obtained as a relict from the construction process (Section 5.3.2). It is the same structure that we use for merging updates (Section 5.3.1).

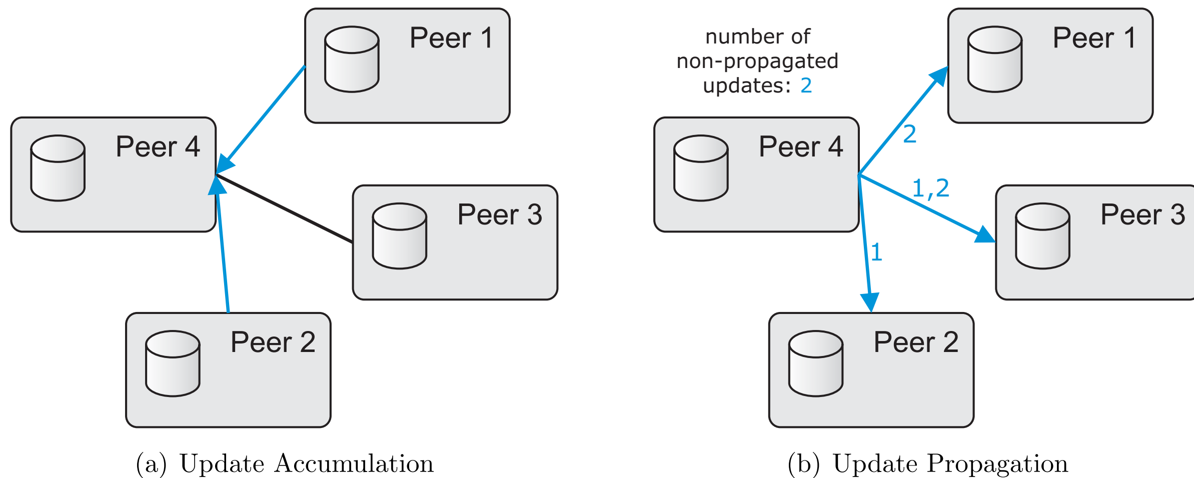
The main advantage of threshold-based strategies in comparison to IPS is the obvious accumulation of updates, which results in a lower number of update messages and thus network load. The disadvantages are higher memory consumption and computational load as well as a higher number of false routing decisions due to the delayed update propagation. In principle, applying threshold-based strategies each peer reacts the same on update events – may they be updates to the local data or updates to the routing index that have been received from a neighbor. The course of action is the same for both threshold-based strategies that we present below. The basic steps are:

1. At first, the updates need to be applied to the routing index or to the local index and the local data.
2. Some measures that the propagation rules may be defined on need to determine to what extent an index has been changed. For this reason, we need to compare the original index to the updated one. Thus, a peer needs a backup of the original version (i.e., the version that existed right before the first non-propagated update arrived) so that the peer can compare later versions to the backup.
3. Updates that affect the same bucket might compensate each other (incrementing and decrementing the bucket count). Such compensating updates can be identified and eliminated.
4. Thereafter, a peer needs to decide whether to propagate its current set of non-propagated updates – consisting of local updates as well as of updates received from neighbors. This decision solely depends on the applied strategy and the propagation rules. In order to reduce network load, it is possible to combine updates into a more compact format before their propagation.
5. Finally, all propagated updates and their corresponding index backups are deleted in order to release memory.

Simple Threshold Propagation Strategy (STPS)

Applying STPS the decision whether to propagate updates or not is made for all neighbors altogether. Thus, when the threshold is exceeded, all neighbors receive updates even

though this means some of them receive only a small number. Figure 5.27 shows an example for STPS – for simplicity neglecting local data updates, i.e., changes to the local index, and using a simple propagation rule, i.e., propagation is triggered once *more than* one update has been accumulated.



(a) Update Accumulation (b) Update Propagation

Figure 5.27: Example for Update Propagation Using the Simple Threshold Propagation Strategy (STPS). *Propagation Rule: Number of Non-Propagated Updates* > 1

In Figure 5.27(a) P_4 receives two updates, one from P_1 and another one from P_2. Not until P_4 has received the second update, does it begin propagation (Figure 5.27(b)) because only then the threshold defined by the propagation rule is exceeded. Updates are propagated to all of P_4's neighbors: P_1 and P_2 each receive one update and P_3 receives two.

The complete workflow of STPS is sketched in Algorithm 13. First, the new updates are incorporated into the index (lines 2–10). As mentioned above, updates are only stored if they have caused a detectable change in the index. Afterwards, the change rate over all indexes (local index and routing indexes) is determined (line 12) and compared to the given threshold. If the threshold is not exceeded, the algorithm ends (line 19). If it is exceeded, the updates are optionally merged (lines 20–24) and then propagated (lines 26–34). Finally, the propagated updates and index copies are deleted from the local storage (lines 36–41).

Advanced Threshold Propagation Strategy (ATPS)

In contrast to STPS, this strategy decides for each neighbor in separate whether to forward updates or not. The advantage is that a neighbor only receives updates if they represent a major change to the current state of its routing index. The disadvantage that comes along is a higher consumption of main memory and increased costs for maintaining backups. To illustrate the difference between ATPS and STPS, the example of Figure 5.28 makes use of the same scenario and the same propagation rule as the example of Figure 5.27.

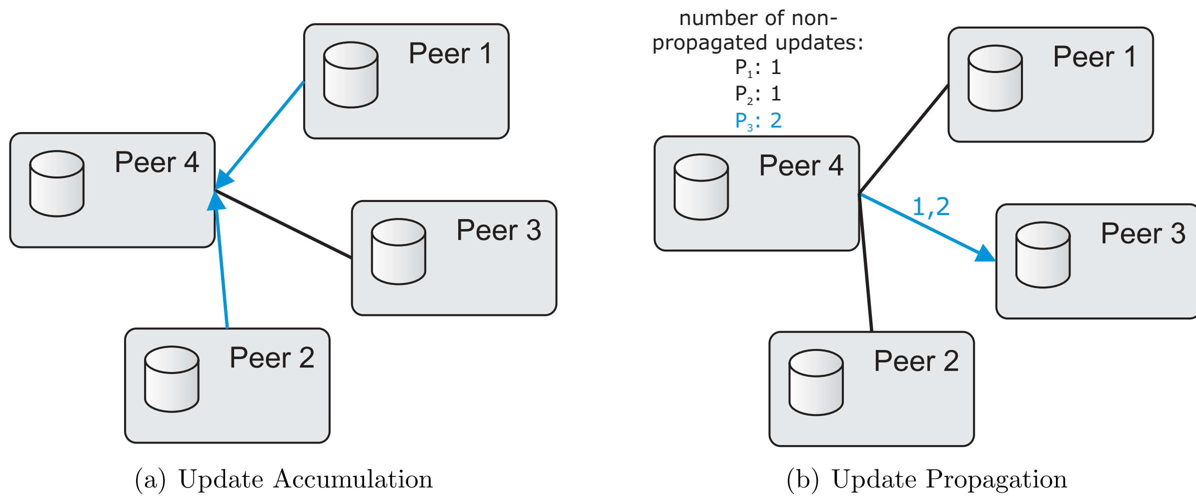
In principle, ATPS uses the same routines as STPS to determine change rates, i.e., the degree of change between the current state of the routing index and a previous state.

Algorithm 13 *processUpdatesSTPS(index, updates)*

```
1: /* 1. update local index or routing index */
2: index.createBackupIfNecessary();
3: oldChangeInfo = index.computeChangeInfo();
4: index.insert(updates);
5: newChangeInfo = index.computeChangeInfo();
6: if newChangeInfo! = oldChangeInfo then
7:    index.store(updates);
8: else
9:    return;
10: end if
11: /* 2. compute change measures for all indexes altogether */
12: totalChangeInfo = computeTotalChangeInfo()
13: /* 3. check if threshold is exceeded */
14: for measure ∈ getMeasures() do
15:    if totalChangeInfo[measure] ≥ measure.getThreshold() then
16:       goto PROPAGATE;
17:    end if
18: end for
19: return;
20: PROPAGATE: /* 4. merging updates */
21: getLI().aggregateUpdates();
22: for neighbor ∈ getNeighbors() do
23:    neighbor.getRI().aggregateUpdates();
24: end for
25: /* 5. propagating updates */
26: for currentNeighbor ∈ getNeighbors() do
27:    updatesToSend = getLI().getRecentUpdates();
28:    for neighbor ∈ getNeighbors() do
29:       if currentNeighbor.getPeerID() != neighbor.getPeerID() then
30:          updatesToSend.add(neighbor.getRI().getRecentUpdates());
31:       end if
32:    end for
33:    currentNeighbor.sendUpdateMessage(updatesToSend);
34: end for
35: /* 6. cleaning up */
36: getLI().clearRecentUpdates();
37: getLI().removeBackup();
38: for neighbor ∈ getNeighbors() do
39:    neighbor.getRI().clearRecentUpdates();
40:    neighbor.getRI().removeBackup();
41: end for
```

Whereas STPS performs this comparison for all neighbors altogether, ATPS performs it for each neighbor in isolation. Hence, each neighbor might receive updates at different times. In contrast to STPS, it is no longer possible to have only one backup of the routing index (in comparison to which the current change rate can be determined) but to have one backup for each neighbor (corresponding to the state of the routing index when the neighbor has last been propagated updates). However, for the simple measure that we use in the example of Figure 5.28, P_4 is able to determine the change rates even without maintaining any backups of its routing index. In this example only the change rate for neighbor P_3 exceeds the given threshold. Thus, only P_3 receives the two updates originating from P_1 and P_2.

Algorithm 14 summarizes the procedure for ATPS in more detail. At first, the set of neighbors that might possibly receive updates is determined (lines 2–3) – if the local index has been changed, this set contains all the peer's neighbors. Afterwards, for each of these neighbors one copy of the indexes is generated if not yet existing (line 5). Then, the updates are inserted into the current version of the index (line 7). Just as for STPS updates are discarded when no changes with respect to the given measures can be detected



(a) Update Accumulation (b) Update Propagation

Figure 5.28: Example for Update Propagation Using the Advanced Threshold Propagation Strategy (ATPS) *Propagation Rule: Number of Non-Propagated Updates* > 1

(lines 10–11). The total change rate (in consideration of both routing index and local index) is determined for each neighbor in isolation using the corresponding index copies and the current version of the index (line 17). The resulting change rates are compared to the corresponding thresholds (lines 19–20). Only if a threshold is exceeded, updates are propagated to the corresponding neighbor (lines 30–36). Before doing so, the updates may be merged (lines 25–28). Finally, all propagated updates are removed and index copies are deleted if possible (lines 38–43).

Measures for Change Detection

So far, we have only considered a very simple propagation rule based on the number of non-propagated updates whereas in the following we discuss more complex measures that can be used as well. It is even possible to combine them so that a peer may use multiple propagation rules. Depending on the configuration either all rules must be fulfilled to trigger propagation or only one. However, in the following we assume that already one propagation rule triggers propagation if not stated otherwise.

Definition 5.3.1 (Propagation Rule, Propagation Measure, Change Rate, Goodness Measure)**.** *A* propagation rule *determines if a peer needs to propagate updates to neighboring peers. It consists of a* propagation measure *and a* threshold*. Such a propagation measure determines the degree of change between two versions of a routing index, in general the current version and a backup version. A propagation rule has the form:*

$$rule(x, y) := \begin{cases} true, & \textit{if } measure(x, y) > threshold \\ false, & \textit{otherwise.} \end{cases}$$

with x and y representing two versions of a routing index and threshold $\in \mathbb{R}_0^+$*. The term* change rate *refers to a concrete computation of a propagation measure given two routing index instances, i.e., $measure(x, y)$. The basic building blocks of a propagation measure are termed* goodness measures *and express comparisons between indexes.*

Algorithm 14 *processUpdatesATPS(index, updates)*

```
1: /* 1. determine neighbors to propagate updates to */
2: neighbors = getNeighbors();
3: neighbors.remove(updates.getSender());
4: /* 2. update local index or routing index */
5: index.createBackupsIfNecessaryFor(neighbors);
6: oldChangeInfos = index.computeChangeInfosFor(neighbors);
7: index.insert(updates);
8: newChangeInfos = index.computeChangeInfosFor(neighbors);
9: for n in neighbors do
10:    if newChangeInfosFor(n) != oldChangeInfosFor(n) then
11:       index.storeUpdatesForNeighbor(n, updates);
12:    end if
13: end for
14: /* for each neighbor in isolation */
15: for n ∈ getNeighbors() do
16:    // 3. compute change measures for all indexes altogether
17:    totalChangeInfo = computeTotalChangeInfoFor(n);
18:    // 4. check if any threshold is exceeded
19:    for measure ∈ localPer.getMeasures() do
20:       if totalChangeInfo[measure] ≥ measure.getThreshold() then
21:          goto PROPAGATE;
22:       end if
23:    end for
24:    /* 5. merging updates */
25:    getLI().aggregateUpdatesFor(n);
26:    for neighbor ∈ getNeighbors() do
27:       neighbor.getRI().aggregateUpdatesFor(n);
28:    end for
29:    /* 6. propagating updates */
30:    updatesToSend = getLI().getRecentUpdatesFor(n);
31:    for neighbor ∈ getNeighbors() do
32:       if n.getPeerID() != neighbor.getPeerID() then
33:          updatesToSend.add(neighbor.getRI().getRecentUpdatesFor(n));
34:       end if
35:    end for
36:    n.sendUpdateMessage(updatesToSend);
37:    /* 7. cleaning up */
38:    getLI().clearRecentUpdatesFor(n);
39:    getLI().removeBackupFor(n);
40:    for neighbor ∈ getNeighbors() do
41:       neighbor.getRI().clearRecentUpdatesFor(n);
42:       neighbor.getRI().removeBackupFor(n);
43:    end for
44: end for
```

We distinguish between three approaches to define propagation measures: (i) *index-level comparison*, (ii) *bucket-level comparison*, and (iii) *index-independent comparison.* Let us begin with measures based on index-level comparisons. They use two goodness measures to determine the change rate: one goodness measure for each of the two routing index versions (current and backup). This kind of goodness measures is computed for each index in isolation, independently from the other. The following basic definitions illustrate some example goodness measures that can be used to define appropriate propagation measures for bucket-based routing indexes such as the QTree and multidimensional histograms. $|B_{old}|$ denotes the number of buckets in the backup copy, $|B_{new}|$ the number of buckets in the current updated version of the index, $|D|$ the number of indexed dimensions (attributes), and *b.low* (*b.high*) the lower (upper) boundary of a bucket b:

- the average extension of all buckets in the old (resp. new) index: $E_{avg,old} = \frac{\sum_{b \in B_{old}} e_{avg}(b)}{|B_{old}|}$, $E_{avg,new} = \frac{\sum_{b \in B_{new}} e_{avg}(b)}{|B_{new}|}$ with $e_{avg}(b) = \frac{\sum_{d \in D}(b.high[d] - b.low[d] + 1)}{|D|}$ denoting the average extension of a bucket b
- the maximum extension of all buckets in the old (resp. new) index: $E_{max,old} = \frac{\sum_{b \in B_{old}} e_{max}(b)}{|B_{old}|}$, $E_{max,new} = \frac{\sum_{b \in B_{new}} e_{max}(b)}{|B_{new}|}$ with $e_{max}(b) = \max_{d \in D} \{b.high[d] - b.low[d] + 1\}$ denoting the max. extension of a bucket b

Examples for index-level propagation measures using these goodness measures are:

- $\mathcal{M}_{E_{avg}} = \frac{\max(E_{avg,new},\ E_{avg,old})}{\min(E_{avg,new},\ E_{avg,old})}$ based on the average bucket extension
- $\mathcal{M}_{E_{max}} = \frac{\max(E_{max,new},\ E_{max,old})}{\min(E_{max,new},\ E_{max,old})}$ based on the maximum bucket extension

As we have stated above, propagation measures need to consider changes to both the routing index and the local data. As we maintain a summary of the local data using the same structure as the summaries that are part of the routing index, a simple solution to the problem is to treat the summary of the local data as part of the routing index. Doing so, we can easily determine the change rate based on both routing index and local data without any adaptations to the propagation measures defined above (and below).

Propagation measures of the second class (based on bucket-level comparisons) do not use goodness measures on index-level but goodness measures on bucket-level that compare different versions of the buckets that a routing index consists of. Thus, the sets of buckets of both index versions are determined (B_{old} and B_{new}) and compared to each other. For this purpose, the index has to provide a means to unambiguously identify corresponding buckets in different index versions, e.g., by applying IDs. If a bucket b is only contained in one of the two sets, then it has either been deleted ($b \in B_{old} \wedge b \notin B_{new}$) or newly created ($b \notin B_{old} \wedge b \in B_{new}$).

Let us again define some helpful goodness measures that provide fundamental definitions, based on which propagation measures using bucket-level comparisons can be defined. $|B_{all}|$ denotes the number of *all* buckets (added, deleted, changed, or unchanged) and $|B_{changed}|$ the number of all buckets that have been affected by updates (added, deleted, or changed). The term *count* refers to the number of records that a bucket represents.

- weighted change in the count of a bucket b:
$$\hat{c}(b) = \begin{cases} \frac{\max(b_{new}.count,\ b_{old}.count)}{\min(b_{new}.count,\ b_{old}.count)}, & \text{if } b \in B_{old} \wedge\ b \in B_{new} \\ b.count, & \text{otherwise} \end{cases}$$
- weighted change in bucket b's average extension:
$$\hat{e}_{avg}(b) = \begin{cases} \frac{\max(e_{avg}(b_{new}),\ e_{avg}(b_{old}))}{\min(e_{avg}(b_{new}),\ e_{avg}(b_{old}))}, & \text{if } b \in B_{old} \wedge\ b \in B_{new} \\ e_{avg}(b), & \text{otherwise} \end{cases}$$

- weighted change in bucket b's maximum extension:
$\hat{e}_{max}(b) = \begin{cases} \frac{\max(e_{max}(b_{new}),\ e_{max}(b_{old}))}{\min(e_{max}(b_{new}),\ e_{max}(b_{old}))}, & \text{if } b \in B_{old} \wedge\ b \in B_{new} \\ e_{max}(b), & \text{otherwise} \end{cases}$

With the help of these goodness measures we can now define some concrete propagation measures based on bucket-level comparisons:

- ratio of the number of changed buckets and the total number of buckets:
$\mathcal{M}_{b_{change}} = \frac{|B_{changed}|}{|B_{all}|}$
- average change based on the weighted changes of the bucket counts:
$\mathcal{M}_{\hat{C}} = \frac{\sum_{b \in B_{all}} \hat{c}(b)}{|B_{all}|}$
- average change based on the average extension of buckets:
$\mathcal{M}_{\hat{E}_{avg}} = \frac{\sum_{b \in B_{all}} \hat{e}_{avg}(b)}{|B_{all}|}$
- average change based on the maximum extension of buckets:
$\mathcal{M}_{\hat{E}_{max}} = \frac{\sum_{b \in B_{all}} \hat{e}_{max}(b)}{|B_{all}|}$

Finally, let us consider the third class of propagation measures (index-independent). The main characteristic is that they are defined independently from the index. We have already used an example of this class to illustrate the principle of STPS and ATPS:

- the number of updates that have not yet been propagated:
$\mathcal{M}_{NonProp} = \#\text{non-propagated updates}$

When selecting propagation measures, it is important to be aware of the fact whether it is possible that the index changes the size of its buckets or not. For example, the region of a QTree bucket can be adapted in its size whereas this is impossible for equi-width multidimensional histograms. In general, the set of propagation rules should be built upon measures that not only concentrate on the size of the buckets but also consider the number of values they represent or the number of non-propagated updates.

5.3.5 Query-Driven Strategies

Strategies of this class update routing indexes triggered by queries. One possibility to implement such a strategy is to send updates piggyback along with query answers. Another one is to use query feedback. In the following, we present two corresponding strategies. The problem that all query-driven strategies have to deal with is that a peer only forwards queries to neighbors that it expects to hold relevant data. But if a peer is not queried with respect to a certain region in the data space, there is no feedback that could be used to update the routing index with respect to the region. Thus, query-driven update strategies should provide a solution to this problem. The advantage of these strategies is that they do not require any backup copies of indexes. They only need to remember which updates have been sent to which neighbors.

Query Estimation Strategy (QES)

Remember the steps of query processing that we have illustrated in Figure 5.1. In the optimization step, a peer needs to determine which of its neighbors provide relevant data. It does so by estimating the size of the result (est_n) for each neighbor n based on the overlap of the queried region with buckets of its routing index describing the data accessible via n – est_n is defined as:

$$\begin{aligned} est_n(range_q) &= \sum_{b \in RI_n} est_b(range_q) \quad \text{with} \\ est_b(range_q) &= \frac{sizeOf(overlap(b, range_q))}{sizeOf(b)} \cdot b.count \end{aligned} \tag{5.12}$$

RI_n denotes the set of buckets in the routing index for neighbor n, $range_q$ the queried region, and $b.count$ the number of elements represented by bucket b. If a neighbor n holds relevant data according to the routing index (i.e., $est_n > 0$) and the query processing strategy (Chapter 6) also identifies n to be relevant, n is queried and the expected number of results est_n is forwarded to n along with the query.

After having computed the answer to the query, the neighbor compares the expected number of records ($res_{est} \widehat{=} est_n$) to the actual number ($res_{act}$). If they differ in a substantial way (i.e., the difference exceeds a threshold), the receiver sends all its updates piggyback along with its answer message. The measure we use to make this decision is defined as:

$$\delta = \frac{\max(res_{act}, res_{est})}{\min(res_{act}, res_{est})} > threshold$$

If the threshold is set to 0, this means that always all available updates are sent along with an answer message. Note that again the set of updates that is forwarded to a particular neighbor has to be determined individually for each neighbor and that in order to reduce network bandwidth the updates can optionally be merged before propagation.

Still, the problem that some portions of a routing index will never be updated remains to be solved. The standard solution we use is to explicitly send update messages to a neighbor if it has not sent a query within a time period of length t_x. Alternatively, we could apply techniques similar to the ones used by the threshold propagation strategies, i.e., active propagation of updates when a certain change rate is exceeded.

The advantage of this strategy is that none respectively only a few explicit update messages have to be sent. It is important to note that even though the indexes might correctly represent the data, the result size estimation might be incorrect due to the approximating characteristic of the summary. A disadvantage of this strategy is the delay that occurs when peers that are not queried frequently update their data – even though they might change their data into a range that is queried frequently. Furthermore, incremental query processing strategies (e.g., IMS, Section 2.5.3) are not applicable in conjunction with this strategy as a peer has to wait until all result records are received to decide whether the threshold is exceeded or not. The problem is that when applying incremental strategies the peer does know when the last message has been received, i.e., it does not know when the answer is complete.

Algorithms 15 and 16 summarize how to adapt query and answer messages when applying QES. Assume the query processing strategy has already determined the set of neighbors (*relevantNeighbors*) that the query is to be forwarded to – in an efficient implementation both tasks (query processing and the QES maintenance strategy) are more closely intertwined so that the result size would be estimated only once. Then, Algorithm 15 is called to add additional information to the query and send the query to the neighbors. Thus, for each neighbor the estimated number of records with respect to the queried region is determined based on the routing index as described above (lines 2–8). If the estimate is greater than 0, it is sent along with the query to the neighbor (lines 10–14).

Algorithm 15 *sendQuery_QES(query, relevantNeighbors)*

```
1: for neighbor ∈ relevantNeighbors do
2:   estimated = 0.0;
3:   buckets = getRIFor(neighbor).getAllBucketsInQueriedSpace(query);
4:   /* 1. estimate result size for current neighbor */
5:   for bucket ∈ buckets do
6:     intersection = intersect(bucket, query);
7:     estimated += intersection.getRatio() * bucket.getCount());
8:   end for
9:   /* 2. query neighbor only if it can contribute */
10:  if estimated > 0.0 then
11:    query.setEstimatedNmbOfResults(estimated);
12:    cache.rememberEstimate(estimated,neighbor,query);
13:    sendQuery(neighbor,query);
14:  end if
15: end for
```

When a peer receives an answer message (Algorithm 16), the updates received along with the answer are applied to the corresponding routing index and stored locally for future propagation to other neighbors (line 3–4). When the last answer message from the peers the query has been forwarded to has been received, the peer tests whether updates need to be sent along with the answer message to the peer that the query has been received from in the first place (lines 6–16). Updates are only attached to the answer message if the error between estimation and result exceeds the configured threshold (lines 13–15) or if the number of result records is 0 (lines 8–10). To reduce network load, updates again can be merged before propagation.

Algorithm 16 *receiveAnswer_QES(answer)*

```
1: neighbor = answer.getSender();
2: /* 1. update routing index according to the received updates */
3: getRIFor(neighbor).applyUpdates(answer.getUpdates());
4: cache.storeUpdates(neighbor, answer.getUpdates());
5: /* 2. if all queried neighbors have answered */
6: if answers have been received now from all queried neighbors and this is not the initiator of the query then
7:   actual = cache.getResults(answer.ID).getCount();
8:   if actual == 0.0 then
9:     attachToOwnAnswer(getNonPropagatedUpdatesForQuerySender(cache.getSenderOfQuery(answer)));
10:  end if
11:  estimated = cache.getEstimatedNmbOfResults(answer.ID,neighbor);
12:  error = max(actual, estimated) / min(actual, estimated);
13:  if error > getThresholdForError() then
14:    attachToOwnAnswer(getNonPropagatedUpdatesForQuerySender(cache.getSenderOfQuery(answer)));
15:  end if
16: end if
```

Query Feedback Strategy (QFS)

In Section 2.4 we have already discussed several approaches towards histogram maintenance using query feedback. Although these approaches work efficiently for histograms in the context of selectivity estimation, they produce histograms that are not applicable, or rather not recommendable, for use as base structures of routing indexes. But we can still use the principle of query feedback to update routing indexes.

In contrast to all strategies discussed so far, there is no situation in which QFS sends explicit update messages. Instead, it only uses the answers to a query to adapt the routing indexes. After having received the answer message to the query from a neighbor n, all buckets of the routing index' summary corresponding to n that overlap the queried region are identified. Then, the algorithm assigns each received answer record to one of the buckets – in this way determining the number of records (act_b) that have been received for a bucket b. Afterwards, in a similar manner as defined above for the query estimation strategy (Equation 5.12), the estimated number of results (est_b) for each of these buckets is determined in consideration of the overlap with the queried region.

If a bucket is completely enclosed by the queried region, it is either deleted if no result records could be assigned to it ($act_b = 0$) or its count is set to act_b. In order to improve approximation quality for the QTree, which works with variable bucket sizes, the old bucket can be removed and a new smaller bucket containing the result records can be inserted instead.

If a bucket only partially overlaps the queried region, the number of records that it represents (*b.count*) is updated. The correction value ($correct_b$) for each such bucket is determined as:

$$correct_b = act_b - est_b$$

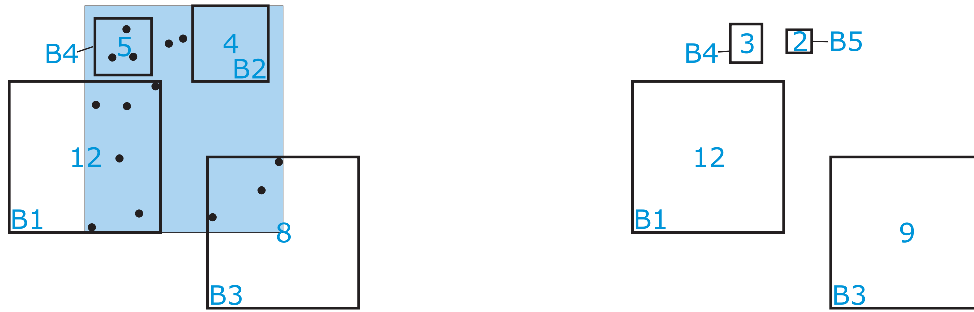
Then, the new number of represented records for bucket b, i.e., a new value for *b.count*, is determined as

$$b.count_{new} = b.count_{old} + correct_b$$

If the resulting new bucket count has a rounded value of 0, the corresponding bucket is deleted. Finally, it is possible that the answer message contains records that cannot be associated with any existing bucket, i.e., records that have been added to the neighbor's data. To update the routing index, these result records are inserted using standard insertion algorithms that have been used for construction.

Figure 5.29 shows an example for updating a summary using query feedback. Figure 5.29(a) depicts buckets (B_1 through B_4), the queried region for a query q (highlighted in blue), and the result records received from the corresponding neighbor. The estimated number of records for each bucket are: $est_{B_1} = \frac{1}{2} \cdot 12 = 6$, $est_{B_2} = 1 \cdot 4 = 4$, $est_{B_3} = \frac{1}{4} \cdot 8 = 2$, $est_{B_4} = 1 \cdot 5 = 5$. Therefore, the total number of expected result records for the corresponding neighbor n is $est_n(q) = 17$. The result set that has actually been received consists of only 14 records.

Using this result set to update the routing index, we compare the estimated number of records per bucket act_b to the number est_b that has actually been received: $act_{B_1} = 6$, $act_{B_2} = 0$, $act_{B_3} = 3$, and $act_{B_4} = 3$. As $act_{B_1} = est_{B_1}$, B_1 is not updated. B_2 is removed because $act_{B_2} = 0$ and it is completely contained in the queried region. Thus, if the records that it represents still existed, they would be part of the result set. B_3's count is updated because the expected number is not the same as the actual number of result



(a) Buckets Before Applying Query Feedback (b) Buckets After Applying Query Feedback

Figure 5.29: Example for Updating Routing Indexes Using Query Feedback – *Queried Region Highlighted in Blue*

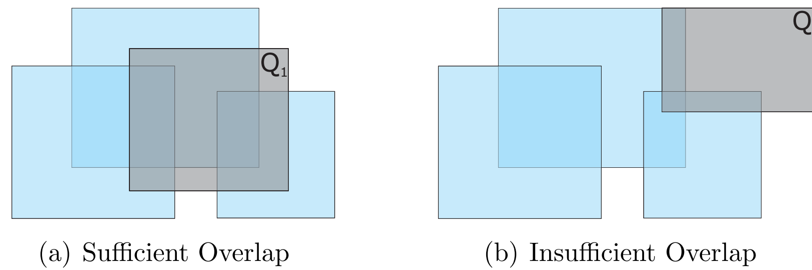
records. B_4 is updated in both, the number of records and its region. And finally, we need to create a new bucket B_5 to capture the two records that are part of the result set but could not be assigned to any existing bucket. Figure 5.29(b) shows the updated summary.

We are aware that this represents only a basic strategy that can be optimized in several ways. For example, so far we do not take any measures to avoid oscillating bucket counts, i.e., for one query the bucket count is increased whereas it is decreased for the next query. Furthermore, the influence of temporary peer crashes should be narrowed down as applying QFS without further adaptations would mean that all summary entries of the temporarily non-available peer would be removed. To simplify matters, we assume in the following that these problems do not occur and leave the improvements to future work.

To solve the problem that a peer is not queried if the routing index does not indicate that it could contribute to the result, QFS maintains a query cache for each neighbor. In this cache the last c queried regions are stored, it only contains queries that are not older than t_p time steps. Queries whose regions are completely enclosed by those of more recent queries are replaced. Before a peer forwards a query to a neighbor, it determines the degree of overlap between the queried region and the regions in the cache – applying the sieve formula [8]. If the overlap γ is too small (i.e., smaller than a threshold of for instance 90%), the query is sent to the corresponding peer – ignoring the information provided by the routing index since it is considered to be out-of-date. A possible extension is to exploit partial results of the cache by determining compensating queries addressing only the non-overlapping regions [44].

Figure 5.30 shows an example query cache with three entries and two queries: Q_1 (Figure 5.30(a)) and Q_2 (Figure 5.30(b)). When processing Q_1, the routing index is considered for making routing decisions because the queried region has a sufficient overlap with the cached regions, i.e., the index is assumed to be up-to-date with respect to the queried region. In contrast, in case of Q_2 the degree of overlap between Q_2 and the cached regions is insufficient. Thus, the information provided by the routing index concerning the currently considered neighbor is not considered for query routing.

The algorithm corresponding to QFS is sketched in Algorithm 17. At first, all buckets that overlap the queried region and correspond to the part of the routing index describing



(a) Sufficient Overlap (b) Insufficient Overlap

Figure 5.30: Exploiting Cached Knowledge about Queried Regions – *Cache Entries (blue), Query (grey)*

the sender's data are identified (lines 2–3). Then, the algorithm iterates through the list of these buckets. First, it determines how many records of the received answer message are contained in the currently considered bucket (lines 5–13). Afterwards, the algorithm determines the intersection ratio of the bucket's region with the queried region (line 15). If a bucket is completely enclosed by the queried region, it is either removed (line 20) if it contains no result records or its count is replaced with the number of contained result records (line 22). In order to improve approximation quality for indexes that do not work with fixed bucket sizes (e.g., the QTree), the old bucket can be removed and a new smaller bucket containing the result records can be inserted instead. If a bucket only partially overlaps the queried region, the number of elements in the overlap has to be estimated (overlap ratio multiplied by bucket count – line 26) and the corresponding error (difference of received and estimated number of records) has to be determined (line 27). Then, a new number of represented records, i.e., a new value for *count*, is computed for the bucket using the estimation and the error (line 28). The new bucket count value is set to the sum of the old count value and the error (line 33) or if the resulting new bucket count has a rounded value of 0, the corresponding bucket is removed (line 31). Finally, all result records that could not be associated with any existing bucket are inserted into the index (line 38).

The main advantage of this strategy is that there is no need to send explicit update messages, which minimizes network load. The application of QFS is particularly useful in situations where the computation of an initial routing index is too expensive or simply impossible. This strategy can also be used as an alternative algorithm to construct routing indexes. However, without any routing indexes the first queries in the network have to be routed to all peers anyway (flooding) in order to obtain the complete result set. Thus, combining such queries with a construction algorithm still is an appropriate alternative and leads to the construction algorithm we have discussed in Section 5.3.2. On the other hand, it is disadvantageous that the accuracy of the routing indexes strongly depends on the number as well as on the queried ranges of issued queries. The need to send queries to neighbors in order to update routing indexes is a direct intervention into the query processing strategy and might complicate the design of an efficient strategy. Besides, the choice of parameters (cache size, maximum age of cache entries, and degree of overlap) is complex and has to be made for each system individually. Of course, we could again have peers send explicit update messages after a predefined time interval, but this would forfeit the advantage of not having to send explicit update messages.

Algorithm 17 *receiveAnswer_QFS(answer)*

```
1: /* 1. identify all buckets that overlap the queried region */
2: index = getRIFor(answer.getSender());
3: buckets = index.getAllBucketsInQueriedSpace(cache.getQuery(answer.ID));
4: results = answer.getResults();
5: for bucket ∈ buckets do
6:    /* 2. determine the number of result elements that lie in the current bucket */
7:    currentCount = 0.0;
8:    for element ∈ results do
9:       if bucket.contains(element) then
10:         currentCount++;
11:         results.remove(element);
12:      end if
13:   end for
14:   /* 3. compute overlap with queried region */
15:   intersectionRatio = intersect(bucket, cache.getQuery(answer.ID));
16:   /* 4a. if bucket is completely enclosed in the queried region */
17:   if intersectionRatio == 1.0 then
18:      /* delete bucket if empty */
19:      if currentCount == 0.0 then
20:         index.remove(bucket);
21:      else
22:         bucket.setCount(currentCount);
23:      end if
24:   else
25:      /* 4b. if bucket is only partially enclosed in the queried region */
26:      estimated += intersectionRatio * bucket.getCount());
27:      error = currentCount − estimated;
28:      newCount = bucket.getCount() + error;
29:      /* remove bucket if the rounded number of records within is 0 */
30:      if newCount < 0.5 then
31:         index.remove(bucket);
32:      else
33:         bucket.setCount(newCount);
34:      end if
35:   end if
36: end for
37: /* 5. insert elements that could not be assigned to any bucket */
38: index.insert(results);
```

Another disadvantage of the query feedback strategy is that sometimes we do not know which buckets to "blame" for the wrong estimation of result records. Assume we have a range query whose range overlaps two buckets corresponding to neighbor n (b_1 and b_2, 50% overlap each), both with a count value of 1. Thus, $est_n = 1$ and $est_{b_1} = est_{b_2} = 0.5$. The neighbor does not send any result records: $act_n = act_{b_1} = act_{b_2} = 0$. QFS would adapt both bucket counts to 0.5 – resulting in non-integer counts. We have shared the "blame" between two buckets although the originally deleted record can only be contained in one of them. Eventually, we can never safely delete buckets because we can never be sure that all records it represents have been deleted. This can easily lead to estimation errors when issuing the next query. We do not have to face these problems using the other strategies because they receive more detailed information about updates.

Both query-driven strategies strongly depend on the queried region, which is crucial to determine both est_n and act_n. Whereas determining the queried region for range queries is straightforward, this is more complicated for top-N and skyline queries. For top-N queries the queried region can be defined as the region that all neighbors have considered to answer the query. An example for such a region is highlighted in Figure 2.1 by the circle that contains all result records. However, if additional pruning information is sent along with the query (depending on the query processing strategy, Section 6.1),

this region might actually become very small. For skyline queries the application of query-driven strategies is even more restricted because it is not only complicated to determine the queried region but also impossible to determine act_{est} since the answer to the query never represents all records within a bucket (only those that do not dominate each other). Thus, the only way of applying update-driven strategies in conjunction with skyline queries is to attach additional information to the query and answer messages. In this case, however, we can just as well attach updates to the answer message, which means applying another strategy for index maintenance.

5.3.6 Comparison

Before going into details on the evaluation of the proposed strategies, let us recapitulate the advantages and disadvantages that we have already identified in the previous sections – they are summarized in Table 5.1.

	IPS	STPS	ATPS	QES	QFS
number of update messages	- -	+ -	+	+	+ +
update data volume	- -	+ -	+	+	+ +
memory consumption	+	-	- -	+	+ +
merging updates	-	+	+	+	-
freshness / completeness	+ +	+	+ -	-	- -

Table 5.1: Advantages and Disadvantages of the Strategies Discussed in this Section

With respect to the update costs, we identify two aspects: the number of update messages and the network load caused by exchanging updates (update data volume). Of course, QFS performs best in this category because it neither encodes nor propagates updates. IPS, on the other hand, performs worst because it propagates each update (without merging them) through the network resulting in a very high routing index freshness and result completeness[3] whereas QFS has deficiencies in this respect because updates are not explicitly encoded, the “wrong” buckets may have been adapted, and routing index freshness strongly depends on query load and query distribution. Memory consumption for QFS is minimal because no information in addition to the routing indexes themselves need to be held. As there are no explicit updates, merging them is not possible/necessary using QFS.

As QES incorporates techniques from both, QFS and IPS, we expect the update costs to be clearly better than IPS but a little worse than QFS. With respect to routing index freshness, QES should be better than QFS (but clearly worse than IPS) because updates are explicitly encoded. As STPS propagates updates to all neighbors once a threshold is exceeded and as ATPS makes this decision for each neighbor in separate, routing index freshness should be slightly better for STPS. As a consequence, update propagation costs should be slightly higher for STPS. With respect to memory consumption, ATPS performs worst because it needs more information about already propagated updates and change rates (for each neighbor) than STPS.

[3]Result completeness strongly depends on routing index freshness as false negative routing decisions based on out-dated routing indexes lead to a loss in completeness.

Because the behavior of the maintenance strategies is influenced by many parameters, it is difficult to give guarantees. Moreover, apart from the strategy's parameters there are many more influencing factors: the update rate, update characteristics (e.g., small changes in attribute values, deletions, or insertions), peer crashes, the data distribution, the number of provided records per peer, the amount of assigned memory space for the routing indexes, the layout of the P2P overlay network and its diameter, the existence and the number of cycles, the number of peers, the query rate, queried ranges, query selectivity, etc. As there are so many of them, we argue that in such a scenario it is hard to provide a complete theoretical analysis without making rigorous assumptions, e.g., on the update rate. Thus, we limit our theoretical analysis to aspects we can guarantee without making such assumptions on the influencing factors.

In case of IPS, the routing index is as up-to-date as possible because updates are propagated as soon as they occur. For all threshold-based update-driven strategies, we can give guarantees based on the applied propagation rules. For example, when we use $\mathcal{M}_{NonProp}$ in conjunction with a threshold of 4. Then, the worst case scenario is that each peer in a network of n peers has collected 3 updates but not yet propagated them. Thus, the maximum number of missed updates at an arbitrary peer is

$$\#\text{missed updates} = (n-1) \cdot (threshold - 1)$$

Similar conclusions can be drawn based on all the other propagation measures proposed in Section 5.3.4.

For query-driven strategies, we cannot derive the same kind of guarantees. But for QES we can distinguish between two categories of neighboring peers: those that have been recently queried and those that have not. For the former we can guarantee that if the deviation between result size estimation and query result exceeded the QES threshold parameter, we have already updated the routing index or if the threshold was not exceeded, the deviation with respect to recently queried regions is smaller than the threshold. For peers that have not been queried recently, we can guarantee that they do not have collected updates for any longer than a time period of t_w (QES parameter) without propagating them. Finally, for QFS we cannot guarantee much because of the problems we have already discussed. We can only guarantee that all those regions of the data space still contained in the query cache, have recently been updated – we cannot guarantee the indexes' freshness.

With respect to update propagation times, we can determine the minimum/maximum update propagation time for one update in a network of n peers:

$$T_{min} = \lfloor \frac{diameter}{2} \rfloor, \quad T_{max} = n$$

Further aspects need to be considered in order to give a good estimation. Whereas the consideration of transfer and processing time is straightforward, an extension to aspects such as query and update rates (to estimate query/update load and delays), the network structure (cycles), data distribution, and all the other aspects mentioned above is a very complex task. However, even if we discussed these extensions in more detail, we cannot make use of the estimation without making rigorous assumptions on the input parameters, e.g., in P2P networks exact details on the current query load with respect to all participating peers are very hard if not impossible to obtain. Thus, we argue that the

effort required to set up a precise equation is better used for an extensive experimental evaluation – the main results of our evaluation are discussed in the following section.

5.3.7 Evaluation

For our evaluation – in that we wanted to experimentally prove the hypotheses (Table 5.1) – we implemented all strategies in SmurfPDMS (Chapter 7) but set the rewriting issue aside, i.e., all peers had the same schema and bidirectional mappings such that rewriting queries was not necessary. We simulated the behavior of an acyclic unstructured P2P system with 100 peers, each holding distributed data summaries based on multidimensional equi-width histograms (10 buckets per indexed dimension) for its neighboring peers. Although we only present our results obtained for equi-width histograms, the general tendencies are the same for the QTree. However, due to overlapping regions, QTree-based routing indexes have additional problems to deal with as we have already pointed out in Section 5.2.6.

The data provided by each peer represents a cluster of 50 four-dimensional data records with attribute values restricted to the interval $[0, 1000]$. All four attributes are indexed, i.e., the histograms use a total number of 10^4 buckets. Clusters are located at random positions in the data space. Each attribute value has a random deviation to the cluster center coordinates of at most 10. We chose clustered data because in general it can be summarized by the routing indexes with low approximation error. Consequently, indexes are very selective and only a small portion of peers is involved in processing a query. Hence, it is the best scenario to examine the effects of out-dated routing indexes. Nevertheless, we have also performed experiments with other setups (number of peers, data distribution, change rate, base structure, etc.) but observed the same tendencies.

To evaluate the influence of dynamic behavior, we simulated updates of the peers' local data. For this purpose, we generated two different data distributions that both adhere to the characteristics stated above. Each peer is assigned one cluster of each distribution. The data distribution slowly changes from the initial distribution to the target distribution: in each time step 1% of the data in the network is changed and in total 25% at the end of the simulation. The average number of updated records per peer during a simulation is the same for all 100 peers. Note that the peers do not completely change their local data but on average 25%, i.e., in the end providing data in 2 clusters. In all our tests we merged updates when possible.

Let us consider the two most important evaluation criteria for update strategies (Table 5.1): costs and benefit/freshness. As a measure for execution costs we chose the additional network load (i.e., update volume in kByte) caused by updating routing indexes. This refers either to the volume of update messages or to the extra volume caused by attaching updates to answer messages. As an appropriate measure for the routing indexes' freshness we chose result completeness (recall) measured by comparing the retrieved query answer to the "best possible" answer. To measure completeness, we defined a set of 9 "control queries". These range queries are defined on two of the four indexed dimensions and partition the two-dimensional queried projection of the data space into 9 quadratic subspaces of equal sizes. These control queries are issued every 10 time steps at a particular peer. Their result set (res_{act} obtained by using the routing indexes in their current state) is compared to the result that would be retrieved if the indexes were

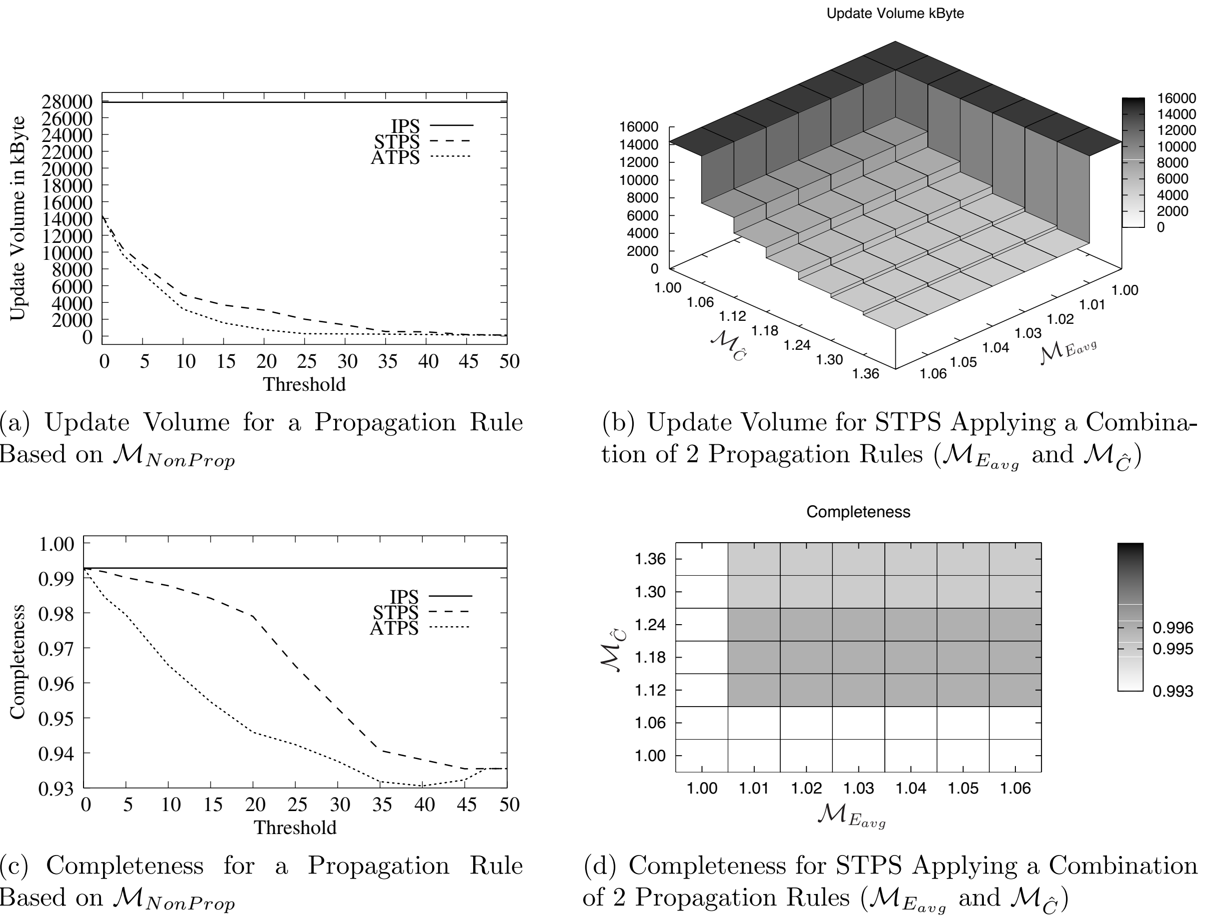
up-to-date (res_{opt} obtained by flooding the network). Completeness is defined as:

$$completeness = \frac{|\{x|x \in res_{act} \wedge x \in res_{opt}\}|}{|res_{opt}|}$$

Note that there cannot exist any $x \in res_{act}$ such that $x \notin res_{opt}$. The completeness values of all figures below refers to the average completeness of all control queries in a test run. In addition, we consider the number of false positive/negative routing decisions displayed as the average values of all issued control queries.

Update-Driven Strategies

Let us begin with the update-driven strategies, Figures 5.31 and 5.32 illustrate the results for our tests using the setup described above. At first, we compared the immediate propagation strategy (IPS) to the threshold strategies (STPS and ATPS). In accordance to the examples that we used to introduce these strategies (Figures 5.27 and 5.28), we first applied $\mathcal{M}_{NonProp}$ (number of non-propagated updates) as propagation measure and varied the threshold from 0 to 50.



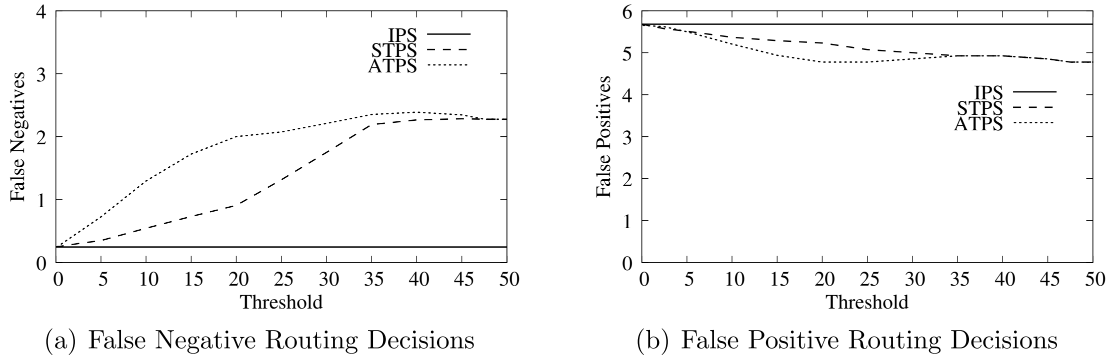
(a) Update Volume for a Propagation Rule Based on $\mathcal{M}_{NonProp}$

(b) Update Volume for STPS Applying a Combination of 2 Propagation Rules ($\mathcal{M}_{E_{avg}}$ and $\mathcal{M}_{\hat{C}}$)

(c) Completeness for a Propagation Rule Based on $\mathcal{M}_{NonProp}$

(d) Completeness for STPS Applying a Combination of 2 Propagation Rules ($\mathcal{M}_{E_{avg}}$ and $\mathcal{M}_{\hat{C}}$)

Figure 5.31: Evaluation Results for Update-Driven Strategies (IPS, STPS, and ATPS)



(a) False Negative Routing Decisions (b) False Positive Routing Decisions

Figure 5.32: False Routing Decisions for Update-Driven Strategies and a Propagation Rule Based on $\mathcal{M}_{NonProp}$

As the network load (update volume) for these three update-driven strategies (Figure 5.31(a)) and the corresponding completeness (Figure. 5.31(c)) show, STPS and ATPS cause a lower network load than IPS but consequently have deficiencies with regard to result completeness because routing indexes are updated less frequently. Although a threshold of 0 means that updates are propagated as soon as they are known, neither IPS nor the threshold strategies achieve a completeness of 1 because in any case it takes some time until the update messages reach all peers whose routing indexes are affected. Even with thresholds of 0 the threshold strategies cause less network load than IPS. The reason is that IPS does not merge updates to a peer's local data.

Applying STPS, updates are propagated to *all* neighbors once the threshold is exceeded for an arbitrary neighbor. Using ATPS the decision is made for each neighbor in separate. Thus, we expected STPS to propagate more updates on average – resulting in a higher network load and a higher completeness (Table 5.1). Fig. 5.31(c) supports our anticipation. Note that in the worst case still 93% of the optimum result set is retrieved. Thus, the difference in performance between STPS and ATPS is low but the tradeoff for STPS between completeness and network load is slightly better than for ATPS because STPS achieves a higher completeness with causing only a little more network load.

Figure 5.32 shows details on false positive and false negative routing decisions. Of course, the number of false negative routing decisions (resulting in missing relevant peers) correlates with the loss in result completeness. IPS results in the lowest number of false negative routing decisions because the very first update indicating that a peer now also provides data in a different cluster than before is propagated through the network right upon performing the first update – threshold propagation strategies using thresholds greater than 0 propagate this information later and so have to deal with a higher number of false negative routing decisions. On the other hand, as IPS sends many update messages, this might cause delays. Thus, in contrast to the other two strategies it sometimes takes longer until the routing indexes correctly represent the information that a peer does no longer provide data within its old region – resulting in false positive routing decisions in the meantime.

We also conducted experiments for combinations of multiple propagation rules – either one of them may alone trigger propagation. Figures 5.31(b) and 5.31(d) show our results for the application of STPS in conjunction with $\mathcal{M}_{E_{avg}}$ (average change in the

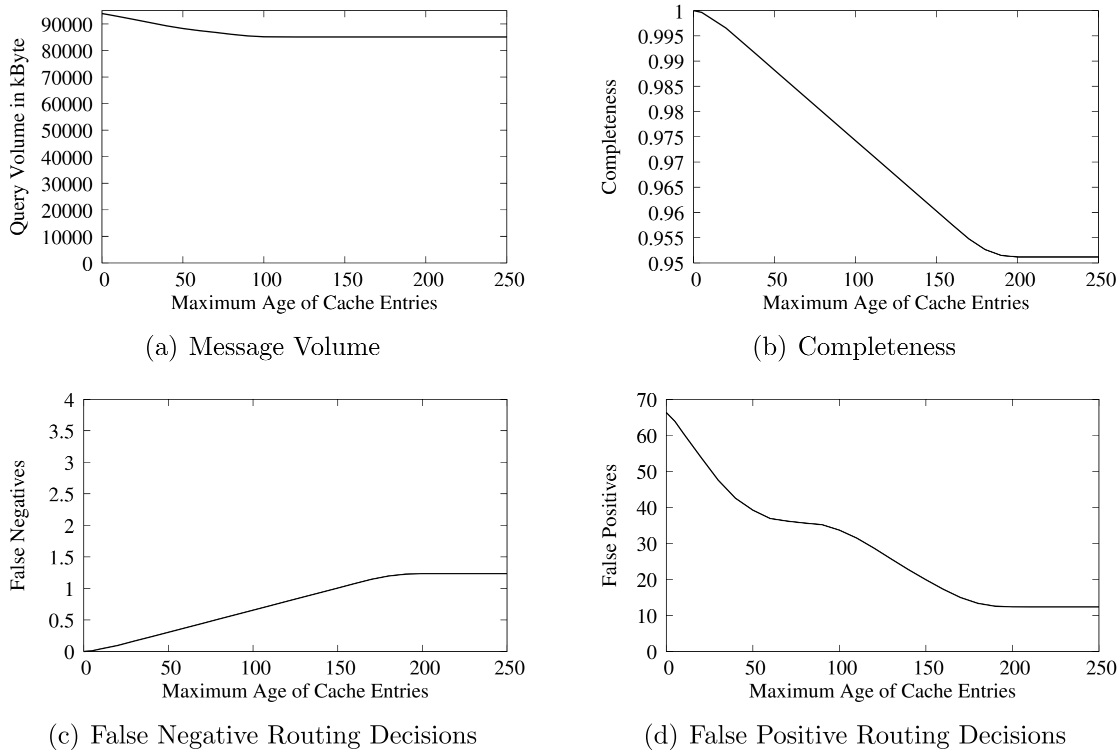
buckets' extensions) and $\mathcal{M}_{\hat{C}}$ (average change in the buckets' counts) – we found the same tendencies applying ATPS. Using a threshold of 1 in conjunction with either one of these two propagation measures means that even if no change has occurred, the propagation rule's condition is always fulfilled. The two figures again show that the higher the thresholds, the less updates are propagated, i.e., the less network load is generated. But they also show another interesting aspect: sending too many updates through the network may cause delays so that routing indexes are not adapted with an optimum time efficiency – thus, completeness may be slightly reduced. The optimal threshold value is big enough to reduce network load but small enough to propagate important updates with little delays. However, the effects in completeness are so small that the general tendency (the higher the thresholds the higher the loss in completeness), which we have identified above, still holds for the average case. As the two figures show, the $\mathcal{M}_{E_{avg}}$ measure does not have any influence on update propagation in our experimental setup. The reason is that the sizes of the regions represented by the buckets in equi-width histograms never change whereas they would if we used a different base structure, e.g., the QTree. These results reveal the importance of taking the structure of routing indexes into account when defining propagation rules.

We also evaluated all other proposed propagation measures. Due to space constraints, we only state the results and omit the presentation of the diagrams. Propagation measures can be ranked according to their sensitivity to updates (descending order): $\mathcal{M}_{NonProp}$, $\mathcal{M}_{b_{change}}$, $\mathcal{M}_{\hat{C}}$, $\mathcal{M}_{\hat{E}_{avg}}$, and $\mathcal{M}_{E_{avg}}$. The more sensitive a measure is, the higher is the resulting network load and the achieved completeness. We found out that measures defined on the maximum bucket extension (i.e., $\mathcal{M}_{\hat{E}_{max}}$ and $\mathcal{M}_{E_{max}}$) should not be used because they only consider the maximum extension of buckets in any dimension. Thus, changes in other dimensions are not detected by this measure. We also tested other data distributions and update patterns and found the same tendencies.

Query-Driven Strategies

The results of our experiments with respect to QFS are shown in Fig. 5.33. As this strategy does not explicitly encode updates, there is no network load caused by propagating updates that we could measure. Instead, QFS increases query load if peers have not been queried recently. However, QFS strongly depends on the frequency of queries as well as on the queried ranges. Thus, in contrast to the previous experiments we needed to issue queries to give the indexes a chance of being updated. We used the 9 control queries as query load and issued them at time steps 0, 10, 20, and 30. The size of the query cache is 9. Since we have only 9 queries in the query load, this means queries are never removed from the cache because of its limited size but only when a newer query with the same region is issued. Thus, we gave QFS optimal premises. We also tested other parametric setups and query loads but in this paper we limit our discussion to the setup introduced above as it represents the best case and all other setups and query patterns may only result in worse results.

For the tests, whose results are shown in Fig. 5.33, we varied the maximum age of cache entries. A maximum age of 0 time steps means there are no cache entries and hence routing indexes are never used. Instead, the network is always flooded in order to process a query – resulting in a very high network load for query processing and a high



(a) Message Volume

(b) Completeness

(c) False Negative Routing Decisions

(d) False Positive Routing Decisions

Figure 5.33: Evaluation Results for QFS

number of false positive routing decisions. By considering routing indexes (max. age of cache entries greater than 0), the number of false positive routing decisions is reduced as the network no longer needs to be flooded for each query. However, by considering routing indexes, it is possible that false negative routing decisions are made if relevant data updates were made in the meantime since the last update.

With cache entry ages of about 200 time steps and more, indexes are only updated once for each range query in the beginning of the simulation. Afterwards, the routing indexes already contain information about the new data distribution. For the rest of the simulation, the cache entries are considered up-to-date with the consequences that indexes are not updated and the result is often incomplete. For lower maximum cache entry ages, entries are discarded during runtime such that queries are sent to additional neighbors than only to those indicated by the routing indexes. Hence, indexes are updated using query feedback resulting in fewer false negative routing decisions.

Fig. 5.34 shows our evaluation results with respect to the application of QES using the same query load as for QFS and varying the threshold for the estimation error, which is responsible for the decision whether to send updates along with a query answer or not. As we have expected, network load as well as completeness decrease with a higher threshold for the approximation error since updates are forwarded less frequently. The update load illustrated in Fig. 5.34(a) is the sum of (i) the volume of updates forwarded along with query messages and (ii) the volume caused by explicit update messages. Fig. 5.34(b) shows the network load caused by query processing in total – this again includes the volume of updates attached to query answers. In comparison to QFS (Fig. 5.33(a)),

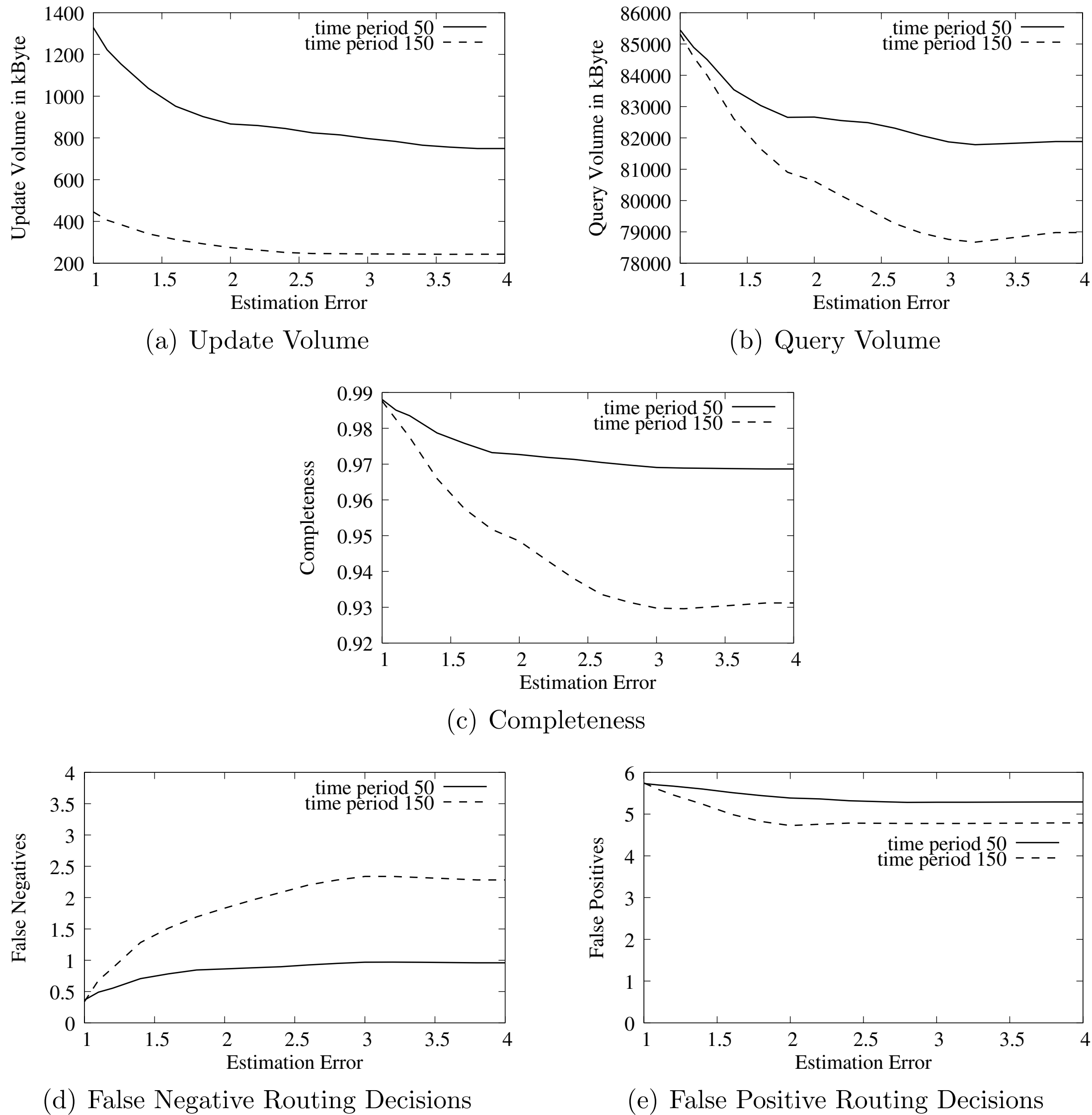


Figure 5.34: Evaluation Results for QES

network load is lower because queries are sent to fewer peers as routing indexes are considered to route queries (supporting Table 5.1). This results in considerably fewer false positive routing decisions. As we have also varied the time period t_w that a peer waits before actively propagating updates with explicit update messages, we see that the lower t_w is, the higher are result completeness, network load, and the lower is the number of false negative routing decisions. However, the number of false positive routing decisions is slightly higher for the lower value of t_w because as routing indexes are updated (reflecting information about the newly available data), more peers are queried and due to the approximation error resulting from summarizing the data, a higher number of queried peers might result in a higher number of false positive routing decisions.

As already mentioned above, we did the same tests with other setups, e.g., network structures, data distributions (e.g., no clusters but data uniformly distributed in the data space), and update rates. We have only presented our results for one setup as they

are representative for the others. In general, however, the network structure determines the minimum costs and benefits that are achievable. The data distribution and the applied base structure for routing indexes determines the quality of the indexes in terms of approximation error – random data, for instance, is harder to summarize than clustered data. In dependence on this quality, a high number of peers might be queried even though the indexes are up-to-date. Thus, the achievable benefit of updating routing indexes is lower. Finally, the update rate (the frequency of updates) influences the offset of benefits and costs, i.e., if more updates occur, more updates need to be propagated and thus maintenance costs increase. Of course, the query pattern is a very important aspect for query-driven strategies. However, in the tests we have discussed above, we gave the strategies optimal premises so that other setups perform worse.

In summary, the test results support our anticipations (Table 5.1). Unfortunately, there is no strategy that performs best with respect to all aspects listed in Table 5.1. So, it still depends on the application and on the user's requirements which strategy to choose. Nevertheless, we conclude that IPS should be applied in relatively static networks, where updates occur only rarely. In such situations the threshold strategies would not propagate the updates soon enough since it may take too much time until the thresholds are exceeded. ATPS should be preferred over STPS in networks with low network bandwidths. Because the application of QFS might lead to failures in the representation of the data by the routing indexes and because QFS strongly depends on the query load, it is only applicable to a small range of applications. QES, on the contrary, propagates updates using explicit update messages and counteracts the problems of QFS. As QES does not solely depend on the query load, it also shows aspects of update-driven propagation strategies. Furthermore, QES can be used in a great variety of applications where defining explicit propagation measures is impossible. Moreover, although not discussed above, it is possible to combine QES with threshold propagation strategies such that updates are actively propagated not only considering a time period but also the accumulation of updates. Thus, in summary, when there are no specific requirements posed by the application and/or exact statistics such as the update rate cannot be obtained, QES is a good choice.

5.4 Conclusion

In this chapter, we defined distributed data summaries (DDSs) as a subclass of routing indexes that fulfill some additional requirements, which make DDSs most beneficial for distributed query processing strategies and systems with dynamic behavior. As our contribution, we presented the QTree as an example base structure for DDSs in addition to multidimensional equi-width histograms and appropriate maintenance strategies for DDSs.

In contrast to histograms that cover the whole data space (defined by the indexed attributes), the QTree only captures subregions that contain data records. Thus, a peer can use the QTree to efficiently identify peers that do not provide relevant data with respect to the query. After having discussed and evaluated some implementational aspects of QTrees, we have also discussed how to keep routing indexes, or rather DDSs, up-to-date. The strategies we have proposed can also be used in conjunction with other base structures. Among these strategies we have identified QFS to be problematic with

respect not only to rank-aware queries but also to the required query load. Thus, we prefer update-driven strategies or QES as they work (almost) independently from the query load. However, if we wanted to use routing indexes specialized on skylines and top-N queries only, we could use more efficient indexes, e.g., based on the approaches recently proposed in [197]. The great disadvantage would be that we would need further indexes to support range queries. Thus, we decided to focus on a more general solution to the problem that can be used for a broad range of query types.

We are aware that the solutions we propose can still be improved in several ways that we plan to investigate in future work. One of them is the analysis whether it is possible to give guarantees for the freshness of routing indexes without making rigorous assumptions. However, this is a difficult problem and might be impossible to solve for the general case. Furthermore, a combination of the strategies could be beneficial, e.g., switching between strategies automatically in dependence on the current query load.

Chapter 6

Query Processing

In the previous chapters we have already discussed the first steps of query processing in PDMSs. We have also already introduced DDSs as well as several strategies to keep them up-to-date. In this chapter, we finally show how all these pieces work together in order to efficiently process rank-aware queries. Note that these techniques are not only useful for PDMSs but can also be applied to unstructured P2P systems in conjunction with DDSs in general. For simplicity we assume the optimizer follows a strategy which plans that a peer waits for all answers from neighbors it has forwarded the query to (QS, Section 2.5.1). Still, the optimizer might just as well follow an incremental strategy (IMS, Section 2.5.3) so that results are forwarded to the initiator as soon as they have been identified. However, implementing such a strategy affects not only step 4 with respect to Figure 6.1 (by identifying relevant neighbors) but also step 6 (postprocessing), which is responsible for the decision of sending answer messages incrementally or not. Besides, in addition to merging the results received from queried neighbors the postprocessing step also includes performing some rewriting tasks that cannot be performed at a neighbor, e.g., computing a join. However, in this chapter we focus on efficient query processing and assume that the remaining operations from rewriting are computed transparently to the query processing strategy.

Although in comparison to a basic flooding approach the mere application of routing indexes already reduces execution costs by far, we consider further options. One of them is the application of rank-aware query operators such as top-N (Definition 2.1.1) and skyline (Definition 2.1.3). Before considering query optimization, we assume a peer to have evaluated the query based on its local data (corresponding to step 3 in Figure 6.1). Hence, we obtain an intermediate result that we can use to prune neighboring peers from consideration. With respect to rank-aware queries, it is for example possible that although a neighbor provides data matching the query predicates, all its data records cannot be ranked better than those of the local result set. As a consequence, this particular neighbor does not have to be queried as none of its records could be part of the final result.

Another possibility to reduce the amount of relevant data in addition to rank-aware query operators is the application of constraints [93]. Assume a user issues a skyline query (Definition 2.1.3) and is not interested in the whole data space, e.g., he/she is not interested in all the libraries but only in those that provide a minimum of $10,000$ books. This kind of restriction conforms to constrained skylines as introduced in Section 2.1.2

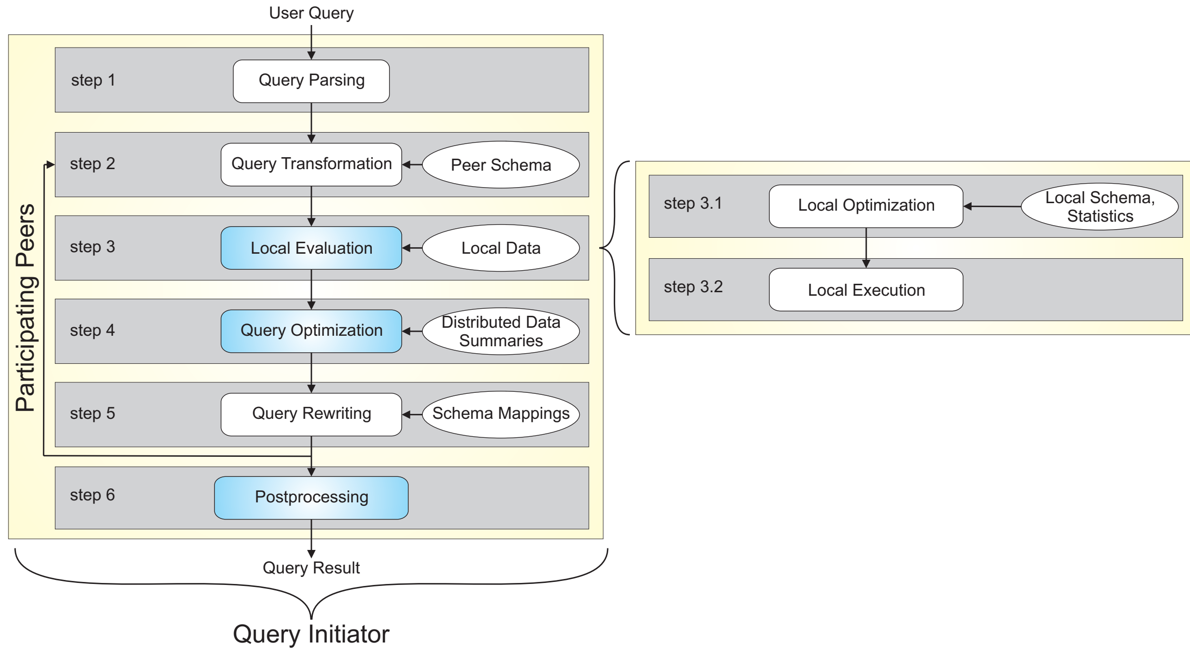


Figure 6.1: Query Processing in PDMSs – Query Optimization

and obviously reduces the relevant data space to only a small portion.

Still, there is another option to reduce the amount of relevant data and thus execution costs: relaxing the completeness/exactness requirements, which are usually posed on the query result, by allowing a controlled amount of relaxation. In this dissertation, inspired by [104], we allow one data record to represent a set of others. By allowing this kind of fuzzy results, query execution costs can be reduced while the user is still provided with the information he/she was looking for. In comparison to the classification of approximation techniques for rank-aware queries of Figure 2.6, our approach [92, 213] is classified as representation-based.

Although this chapter focuses on how to process top-N and skyline queries efficiently, note that our techniques also support simple selection queries, queries involving joins, and all queries that can be formulated on the set of algebra operators presented in Section 3.3. The basic principle for all queries is the same: we exploit the information provided by the POPs that the query tree consists of in conjunction with the information provided by the routing indexes. For select POPs, for example, we can extract constraints and path expressions and identify which neighbors provide relevant data – in case of a QTree-based routing index using the lookup algorithm sketched in Section 5.2.3. The lookup algorithm can also estimate the number of records provided by a neighbor in consideration of the queried region and the buckets of the QTree/histogram that summarizes the neighbor's data. If a neighbor does not provide any relevant data, we can safely prune it from consideration as it cannot contribute to the result. Based on these considerations on the select POPs, we can make some additional considerations for other operators that might be contained in a query. Additional select POPs on higher levels of the query tree, for example, represent additional restrictions of the query space that might help identify irrelevant neighbors. Let us consider a join POP with two select POPs underneath. Due to the above considerations, we have information about what data is provided by a

neighbor with respect to the underlying select POPs. As the buckets of a histogram or a QTree describe the distribution of the records' attribute values, we can even estimate the result size of the join. It is also possible that the attribute values are so different that there cannot be any join result. Thus, in this case neighbors can be pruned because they cannot contribute to the result.

Likewise, we can exploit the special characteristics of other POPs to determine the relevance of neighboring peers. As we will discuss in this chapter in more detail, for top-N and skyline queries we can exploit the pruning characteristic of the ranking functions. Thus, in the following we first introduce the baseline algorithms for processing top-N and skyline queries, which consider a peer's local intermediate result as well as the information provided by DDSs. Then, we extend these algorithms to support constraints and finally extend them once more to support relaxation as well.

6.1 Processing Top-N Queries Using DDSs

In distributed environments query execution costs are only to a small extent determined by local execution factors such as IO/CPU costs or main memory consumption but mainly by the number of peers that are involved in answering a query. Thus, the main goal of an efficient query processing strategy is to enable a peer to identify neighbors providing relevant data with respect to a particular query. The question is how to identify such neighbors without querying them. We have already learned that routing indexes, or DDSs respectively, provide a solution to this problem as they summarize the data accessible via neighboring peers. We have already illustrated how to use this knowledge for selection and range queries in Section 5.2.3 with the QTree as an example structure.

As top-N queries are based on ranking functions, it is not straightforward how to use DDSs to identify relevant neighbors. Thus, we first describe how to achieve this goal before presenting an appropriate algorithm for distributed top-N query processing. Without loss of generality, we assume that the information provided by the DDSs can be represented as *regions* being defined by a data subspace, a number of represented records, and a corresponding neighbor peer.

6.1.1 Top-N Queries on Regions

A peer processing a top-N query uses its local data $\mathcal{B}_{rec}$ (or rather its local intermediate result $\mathcal{B}_{local} \subseteq \mathcal{B}_{rec}$) and the information provided by its routing index as input ($\mathcal{B}_{RI}$). This means the input consists of (i) a set of regions $\mathcal{B}_{RI}$ and (ii) a set of records $\mathcal{B}_{local}$ that can be generalized and treated as regions without extensions. Based on this input ($\mathcal{B}_{in} = \mathcal{B}_{RI} \cup \mathcal{B}_{local}$) the peer needs to identify which neighbors provide relevant data. As each DDS region is assigned an ID that identifies a particular neighbor, the main idea is to decide which regions provide relevant data.

On a set of records, a top-N query is defined by a ranking function r that assigns a score $r(p)$ to each record p. This score is used to rank the records in ascending (MIN annotation) or descending (MAX annotation) order. The result set consists of those N records with the lowest (MIN) or highest (MAX) scores. Without loss of generality, we assume in the following that the top-N query asks for those N records with the lowest

scores with respect to ranking function r (MIN annotation). Whereas it is straightforward to determine the score for a record, a region represents a set of records and thus a set of scores. But we nevertheless need to find a ranking of regions in order to identify those that are relevant to the query. For each region $B \in \mathcal{B}_{in}$ we can determine a lowest score $s_{min}(B)$ and a highest score $s_{max}(B)$ denoting the lowest and highest score values that any record $p \in B$ with respect to r might have. Figure 6.2 illustrates this issue with an example, in which p_{min} and p_{max} represent fictitious records that would be scored lowest and highest (s_{min}, s_{max}) with respect to the query. Using these fictitious records does not mean we assume that they actually exist, their attribute values are determined based on the region's boundaries.

In order to identify the set of regions that are relevant to answer the query, we begin with those N elements of $\mathcal{B}_{rec}$ with the lowest scores (MIN annotation), i.e., we begin with the local answer to the query $\mathcal{B}_{local}$ that we have received as input. The score s_{local} of the record with the highest score (MIN annotation) that is still part of the local result is defined as:

$$s_{local} = \max_{B \in \mathcal{B}_{local}} s_{max}(B) = \max_{B \in \mathcal{B}_{local}} r(B)$$

Based on s_{local} we can determine the set of relevant DDS regions. For this purpose, we first compute $s_{min}(B)$ (MIN annotation) for each DDS region $B \in \mathcal{B}_{RI}$ with respect to ranking function r. We only need to consider all regions $B \in \mathcal{B}_{RI}$ for which $s_{min}(B) \leq s_{local}$ holds (MIN annotation) because only they can represent records that might be ranked better than any record of the local result B_{local}. Then, these regions are ranked together with the records contained in $\mathcal{B}_{local}$ according to their s_{max} values in ascending order (MIN annotation). In case of a record p, s_{min} and s_{max} have the same value: $s_{min}(p) = s_{max}(p) = r(p)$.

The set of regions that provides a sufficient number of records to answer the query $\mathcal{B}_{suff} \subseteq \mathcal{B}_{in}$ is determined by traversing the ranked list of regions and records until all yet visited regions altogether provide at least N records such that the following statement holds:

$$\sum_{B \in \mathcal{B}_{suff}} B.count \geq N\ ,\ s_{worst} := \max_{B \in \mathcal{B}_{suff}} s_{max}(B) \tag{6.1}$$

s_{worst} denotes the highest (MIN annotation) possible score of a record that might be represented by any region contained in $\mathcal{B}_{suff}$, p_{worst} denotes the corresponding fictitious record, i.e., $r(p_{worst}) = s_{worst}$. Based on s_{worst} we can determine all regions $\mathcal{B}_{add} \subseteq \mathcal{B}_{RI} \setminus \mathcal{B}_{suff}$ that might represent records with a score smaller than s_{worst} (MIN annotation), i.e.:

$$\mathcal{B}_{add} := \{B \in \mathcal{B}_{RI} \setminus \mathcal{B}_{suff} |\, s_{min}(B) < s_{worst}\} \tag{6.2}$$

Finally, the peer determines the set of relevant regions $\mathcal{B}_{topN}$ (containing records and regions) as:

$$\mathcal{B}_{topN} := \mathcal{B}_{suff} \cup \mathcal{B}_{add} \tag{6.3}$$

All peers that do not provide data corresponding to any region in $\mathcal{B}_{topN}$ are irrelevant and do not need to participate in answering the query.

Figure 6.2 shows an example scenario in which the user is looking for the top 10 records located closest to the asterisk. In this example, $\mathcal{B}_{suff}$ consists of the grey shaded regions (3 local data records and 2 DDS regions) that altogether represent 10 records.

The worst score that any of these regions might provide with regard to the query is determined by the fictitious record p_{worst}. It defines the maximum distance s_{worst} of the worst record that would be part of the result if only $\mathcal{B}_{suff}$ was considered to answer the query. There are regions that might represent records scored better than p_{worst}. $\mathcal{B}_{add}$ is the set of these regions. In Figure 6.2 they can easily be identified because they overlap the circle around the asterisk defined by p_{worst}. Thus, in the example $\mathcal{B}_{add}$ consists of only one region highlighted in blue. Only those peers that provide data for regions contained in $\mathcal{B}_{topN} = \mathcal{B}_{suff} \cup \mathcal{B}_{add}$ have to be queried.

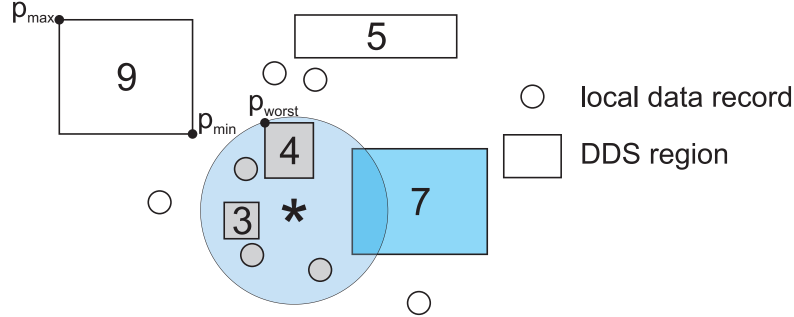


Figure 6.2: Top-N Query Evaluation on Regions. *Query asking for the top* 10 *records located closest to the asterisk*

6.1.2 Distributed Processing of Top-N Queries

Based on the definition of top-N queries on regions, each peer follows the same basic steps to process the query:

- compute the top-N query result on regions considering the local data, DDS regions, and additional information received along with the query,
- forward a top-K query with additional information to each neighboring peer p that provides data of regions in $\mathcal{B}_{topN}$, with

$$K_p := \min \left\{ N, \sum_{B \in \mathcal{B}_{topN},\ B.\text{ID=p}} B.count \right\}, \text{ and} \tag{6.4}$$

- receive all answers and combine the results into a preliminary top-N result, which is either displayed to the user or sent to the peer the query has been received from.

Based on the definition of top-N queries on regions, Algorithm 18 shows how these steps can be implemented. Peers use caches to remember intermediate results and distinguish between query and answer messages. The algorithm consists of three parts: (i) local query processing at the query initiator, (ii) local query processing at a peer that receives a top-N query from a neighbor, and (iii) processing answer messages.

Local query processing at the query's initiator is straightforward and follows the evaluation of a top-N query on regions, which we have discussed in Section 6.1.1. This

Algorithm 18 *Top-N Query Processing*

1: **I. initiate a top-N query**
2: s_{worst},$\mathcal{B}_{topN}$ = topN($\mathcal{B}_{RI}$, $\mathcal{B}_{local}$);
3: askNeighborsTopN($\mathcal{B}_{topN}$,s_{worst},$\mathcal{B}_{RI}$);
4: self.*cache*.storeResult($\mathcal{B}_{topN} \cap \mathcal{B}_{rec}$);
5:
6: **II. receive a top-N query ($s_{worst_{sender}}$)**
7: $\mathcal{B}_{RI} = \{B \in \mathcal{B}_{RI} | B$ not provided by *sender*$\}$;
8: $\mathcal{B}_{RI}$ = removeEntries($\mathcal{B}_{RI}$,$s_{worst_{sender}}$);
9: $\mathcal{B}_{local}$ = removeEntries($\mathcal{B}_{local}$,$s_{worst_{sender}}$);
10: s_{worst},$\mathcal{B}_{topN}$ = topN($\mathcal{B}_{RI}$, $\mathcal{B}_{local}$);
11: s_{worst} = getBestScore(s_{worst}, $s_{worst_{sender}}$);
12: askNeighborsTopN($\mathcal{B}_{topN}$,s_{worst},$\mathcal{B}_{RI}$);
13: self.*cache*.storeResult($\mathcal{B}_{topN} \cap \mathcal{B}_{rec}$);
14:
15: **III. receive a top-N answer ($\mathcal{B}_{answer}$)**
16: $\mathcal{B}_{topN}$ = topN(self.*cache*.getResult() $\cup$ $\mathcal{B}_{answer}$);
17: **if** all queried neighbors have answered **then**
18:   **if** self is initiator **then**
19:     outputResultToTheUser($\mathcal{B}_{topN}$);
20:   **else**
21:     sendTopNAnswer($\mathcal{B}_{topN}$);
22:   **end if**
23: **else**
24:   self.*cache*.storeResult($\mathcal{B}_{topN}$);
25: **end if**

means that the peer processes the query locally based on its local result and the DDS regions, it forwards the query to those neighbors that have been identified as relevant, and it stores the intermediate result $\mathcal{B}_{topN}$ without DDS regions into its local cache. Algorithm 19 provides more details on how to identify and query relevant neighbors given the result of the local computation $\mathcal{B}_{topN}$. In case there are no relevant neighbors, the initiator outputs $\mathcal{B}_{topN}$ to the user as the answer to the query. A query that is forwarded to neighboring peers is adapted as described above (K in Equation 6.4) and attached the score of p_{worst}, i.e., s_{worst}.

Algorithm 19 *askNeighborsTopN(B_{topN}, s_{worst}, $\mathcal{B}_{RI}$)*

1: $\mathcal{P}_{ask} = \{n \in$ self.*neighbors* $| \exists\, a \in \mathcal{B}_{topN} \cap \mathcal{B}_{RI} \quad \wedge \quad a \in$ self.*index*.*getRegionsFor*(n)$\}$;
2: **if** $\mathcal{P}_{ask} = \emptyset$ **then**
3:   **if** self is initiator **then**
4:     outputFinalResult(self.*cache*.getResult());
5:   **else**
6:     sendAnswer(self.*cache*.getResult());
7:   **end if**
8: **else**
9:   **for all** $n \in \mathcal{P}_{ask}$ **do**
10:     $K_n = \min\left\{N, \sum_{B \,\in\, \mathcal{B}_{topN},\; B.\text{ID=p}} B.count\right\}$;
11:     forwardQuery(n, K_n, s_{worst});
12:   **end for**
13: **end if**

Part II of Algorithm 18 summarizes how any peer in the network reacts upon receiving a top-N query from another peer. In contrast to the initiator, such a peer has to ignore all regions of its DDS describing the data of the query's sender. Furthermore, the peer can ignore all local data records and DDS regions that may not provide records with a "better" score than $s_{worst_{sender}}$ (received along with the query). The reason is that all records provided by regions with s_{min} greater than $s_{worst_{sender}}$ (MIN annotation), would be pruned by the sender of the query once it receives them. After removing all these records

and regions, the peer computes the top-N result on regions based on the remaining data (Section 6.1.1). Thus, the peer obtains a result set $\mathcal{B}_{topN}$ of local records and regions and a new highest (MIN annotation) record s_{worst}. By using again Algorithm 19, $MIN(s_{worst}, s_{worst_{sender}})$ (MIN annotation) is forwarded along with the query to all those neighbors providing data in regions contained in $\mathcal{B}_{topN}$. Finally, the intermediate result is stored into the cache.

When a peer receives an answer message (part III of Algorithm 18), it recomputes the top-N query based on the received result and the data that has been stored into the cache. This result is either again stored into the cache or, in case all queried neighbors have already answered, forwarded to the sender of the query. Finally, the initiator receives all answers from these neighbors that it has forwarded the query to, computes the final result, and outputs it to the user.

6.2 Skyline Query Processing

Similar to the previous section we first define the skyline on regions, which allows us to consider DDS information for processing skyline queries. Afterwards, we sketch a distributed query processing strategy that exploits this concept and aims at minimizing the number of involved peers and thus execution costs.

6.2.1 Skylines on Regions

Remember the definition of skyline queries and the dominance relation $\prec_{\mathcal{R}}$ (Definition 2.1.3). The decisive characteristic of a skyline is that it consists of only those records that are not dominated by any other record. A record p dominates another record q with respect to the set of ranking functions $\mathcal{R}$ that the skyline is defined on ($p \prec_{\mathcal{R}} q$) if p is ranked as good as or better than q in all dimensions and ranked better than q in at least one dimension – each dimension corresponds to a ranking function. However, so far this relation is only defined on records. In order to identify relevant DDS regions and neighbors for skyline computation, we need to generalize $\prec_{\mathcal{R}}$ in a way that makes it applicable to regions as well. Intuitively, a region B_1 dominates another region B_2 if all records $p \in B_1$ dominate all records $q \in B_2$. Thus, an appropriate generalization is $\prec_{\mathcal{R}_{Reg}}$:

$$B_1 \prec_{\mathcal{R}_{Reg}} B_2 \Leftrightarrow p \prec_{\mathcal{R}} q \;\forall\, p \in B_1, \forall\, q \in B_2 \tag{6.5}$$

Since data records can be considered as regions without extensions, $\prec_{\mathcal{R}_{Reg}}$ can be applied to regions, records as well as to their combination. As we do not know the exact set of records represented by a DDS region, we still need to find an efficient way to decide whether a region is dominated or not. For this purpose, we again consider two fictitious records (p_{min} and p_{max}) with respect to the set of ranking functions $\mathcal{R}$ – attribute values are again determined based on the region's boundaries. For region B, p_{min} and p_{max} are defined as the two fictitious records for that

$$\forall p \in B, p \neq p_{max}(B) : p \prec_{\mathcal{R}} p_{max}(B) \quad \text{and} \quad \forall q \in B, q \neq p_{min}(B) : p_{min}(B) \prec_{\mathcal{R}} q \tag{6.6}$$

holds. Figure 6.3 shows an example on a two-dimensional data set. In this example and in the following we assume that all ranking functions are annotated with MIN.

Based on p_{min} and p_{max} we can efficiently determine whether a region dominates another one. A region B_1 dominates a region B_2 if and only if $p_{max}(B_1)$ dominates $p_{min}(B_2)$:

$$B_1 \prec_{\mathcal{R}_{Reg}} B_2 \Leftrightarrow p_{max}(B_1) \prec_{\mathcal{R}} p_{min}(B_2) \tag{6.7}$$

Thus, in oder to identify relevant DDS regions, we need to evaluate $\prec_{\mathcal{R}_{Reg}}$ on the union of the local intermediate result $\mathcal{B}_{local}$ (obtained by computing the skyline based on the peer's local data $\mathcal{B}_{rec}$) and the DDS regions $\mathcal{B}_{RI}$. This result, in comparison to the result $\mathcal{B}_{globalSkyline}$ that we would obtain if we evaluated $\prec_{\mathcal{R}}$ directly on the data of all the peers in the network, contains all regions that might represent any record which would be element of $\mathcal{B}_{globalSkyline}$. This means that the evaluation principle introduced above guarantees that (i) each region B_1 which could represent a record $p \in \mathcal{B}_{globalSkyline}$ is contained in the skyline on regions and (ii) there does not exist any region or record B_2 dominating B_1.

Assuming we have two regions B_1 and B_2, B_1 dominating B_2 ($B_1 \prec_{\mathcal{R}_{Reg}} B_2$), each region represents at least one record, i.e., there exists at least one record $p \in B_1$ and at least one record $q \in B_2$. We have determined the dominance $B_1 \prec_{\mathcal{R}_{Reg}} B_2$ by using $p_{max}(B_1)$ and $p_{min}(B_2)$, i.e., $p_{max}(B_1) \prec_{\mathcal{R}} p_{min}(B_2)$. $p_{min}(B_2)$ by definition (Equation 6.6) dominates all records $q \in B_2$ and $p_{max}(B_1)$ is dominated by all records $p \in B_1$, i.e., $\forall p \in B_1, p \neq p_{max}(B_1) : p \prec_{\mathcal{R}} p_{max}(B_1)$ and $\forall q \in B_2, q \neq p_{min}(B_2) : p_{min}(B_2) \prec_{\mathcal{R}} q$. Thus, $\forall p \in B_1, p \neq p_{max}(B_1), \forall q \in B_2, q \neq p_{min}(B_2) : p \prec_{\mathcal{R}} p_{max}(B_1) \prec_{\mathcal{R}} p_{min}(B_2) \prec_{\mathcal{R}} q$. It follows that $\forall p \in B_1, \forall q \in B_2 : p \prec_{\mathcal{R}} q$. Hence, there cannot exist any record $q \in B_2$ that could be part of the skyline.

In summary, the result of a skyline on regions with $\mathcal{B}_{in} = \mathcal{B}_{RI} \cup \mathcal{B}_{local}$ as input is $\mathcal{B}_{sky}$:

$$\mathcal{B}_{sky} := \left\{ B_1 \in \mathcal{B}_{in} | \nexists B_2 \in \mathcal{B}_{in} : B_2 \prec_{\mathcal{R}_{Reg}} B_1 \right\} \tag{6.8}$$

In order to answer the query correctly, the query has to be forwarded only to those neighbors that provide data for any region $B \in \mathcal{B}_{sky}$.

Let us again consider a two-dimensional example (Figure 6.3). We have several local data records and DDS regions. The skyline query is defined on both dimensions and the ranking functions are annotated with MIN. For region B_9, Figure 6.3 illustrates p_{min} and p_{max} with respect to the skyline definition. All records and regions that are part of the skyline are highlighted in grey. Note that B_3 dominates both B_4 and B_5 but not B_7, i.e., $B_3 \prec B_4$, $B_3 \prec B_5$, and $B_3 \not\prec B_7$. Furthermore, B_7 is neither dominated by B_3 nor by B_4 but by a local data record. After the evaluation of the skyline on regions, only B_3 and B_9 have been identified as being relevant to answer the query. Thus, only neighbors that provide data concerning these two regions need to be queried.

As a final remark on the skyline on regions, let us consider what happens if we cannot determine p_{min} and p_{max} easily or unambiguously. For the DDS variants and ranking functions we have discussed so far, regions are always rectangular in shape in terms of the data space defined by the query's ranking functions. This is a handy characteristic since for these regions we can always determine p_{min} and p_{max} unambiguously as defined in Equation 6.6. However, as we allow ranking functions to be defined on multiple attributes, it is possible that (e.g., due to dimension reduction) a region is no longer rectangular in shape. Such regions might represent multiple records that together dominate all other records (which are possibly represented by the region) but not each other – in contrast,

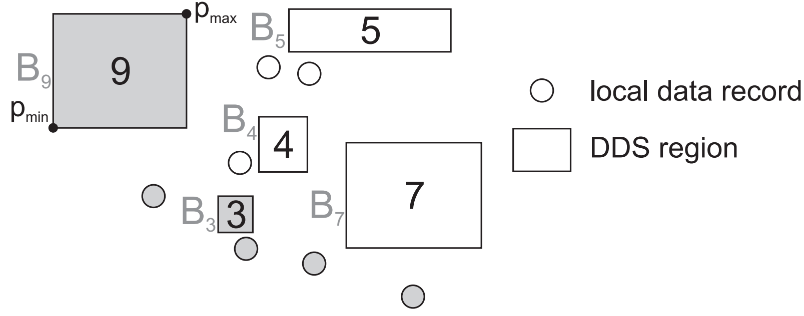


Figure 6.3: Skyline on Regions. *Skyline asking for records that minimize both dimensions*

for rectangular shapes and basic ranking functions there is always only one such fictitious record. In this case we need to represent a region either by multiple p_{min} and p_{max} records or use virtual records.

Assuming we have discovered that in consideration of the ranking functions a region B possibly represents two records that do not dominate each other ($p_{min_1}, p_{min_2} \in B$, $p_{min_1} \not\prec_{\mathcal{R}} p_{min_2}$, $p_{min_2} \not\prec_{\mathcal{R}} p_{min_1}$) but together dominate all other records represented by B ($\forall p \in B, p \notin \{p_{min_1}, p_{min_2}\} : p_{min_1} \prec_{\mathcal{R}} p \vee p_{min_2} \prec_{\mathcal{R}} p$), then we could still use a $p_{min} \notin B$ such that $p_{min} \prec_{\mathcal{R}} p_{min_1} \wedge p_{min} \prec_{\mathcal{R}} p_{min_2}$ to represent the region so that the above considerations for computing the skyline on regions hold even for the general case. These considerations can easily be extended to arbitrary regions with arbitrary numbers of records that do not dominate each other: $p_{min_1}, p_{min_2}, \ldots, p_{min_n}$, $n \in \mathbb{N}$. The same considerations also hold for p_{max} so that given the sets $p_{min_1}, p_{min_2}, \ldots, p_{min_n}$ and $p_{max_1}, p_{max_2}, \ldots, p_{max_m}$ we can define p_{min} based on the set of ranking functions $\mathcal{R}$ as the fictitious record for that

$$\forall r \in \mathcal{R} : r(p_{min}) = \min_{i=1\ldots n} \{r(p_{min_i})\} \tag{6.9}$$

holds and p_{max} as the fictitious record for that

$$\forall r \in \mathcal{R} : r(p_{max}) = \max_{i=1\ldots m} \{r(p_{max_i})\} \tag{6.10}$$

holds. Based on these definitions, we can use the algorithms introduced so far also for arbitrary regions and ranking functions.

6.2.2 Distributed Processing of Skyline Queries

Based on the generalized dominance relation introduced above, we can define an algorithm that each peer adheres to when processing a skyline query. The main steps for each peer are:

- computing the skyline in consideration of the local data, all regions provided by the DDS, and additional information, which might have been received along with the query,
- forwarding the query to all neighboring peers that provide data in regions which are part of the result computed in the previous step, sending additional information along with the query that might help prune more neighbors from consideration, and

- receiving all answers and combining the results into a preliminary skyline, which is either output to the user or sent to the peer the query has been received from.

Just like the algorithm for processing top-N queries, this algorithm again uses two kinds of messages (query and answer messages) and each peer maintains a cache to manage intermediate results. We again distinguish between three parts: (i) a peer initiates a query, (ii) a peer receives a query from a neighbor, and (iii) a peer receives an answer message from one of its neighbors. Algorithm 20 sketches this procedure in more detail.

At first (part I), the initiator computes the skyline result $\mathcal{B}_{sky}$ on the union of its local result $\mathcal{B}_{local}$ and the information provided by its routing index $\mathcal{B}_{RI}$ – using the generalized dominance relation $\prec_{\mathcal{R}_{Reg}}$ of Section 6.2.1. Afterwards, the initiator forwards the query to neighbors providing relevant data, i.e., those peers that provide data for any region contained in $\mathcal{B}_{sky}$. Finally, the local records contained in $\mathcal{B}_{sky}$ are stored as an intermediate result into the cache ($\mathcal{B}_{sky} \cap \mathcal{B}_{rec}$). Algorithm 21 shows more details on how to query neighbors. It works analogously to Algorithm 19 for top-N queries. The main difference is the kind of information that is forwarded along with the query. In case of skyline queries, this is the set of records contained in $\mathcal{B}_{sky}$, i.e., $\mathcal{B}_{sky} \setminus \mathcal{B}_{RI}$. The ranking scores of these records might help to reduce execution costs – the initiator might already know local records dominating a large part of the data space. Not forwarding the information about the dominated space along with the query means the receiver does not know that it can safely neglect some of its neighbors which only provide data in the dominated subspace. Thus, peers would be queried whose result records would be pruned later on.

Algorithm 20 *Skyline Query Processing*

1: **I. initiate a skyline query**
2: $\mathcal{B}_{sky}$ = skyline($\mathcal{B}_{RI} \cup \mathcal{B}_{local}$);
3: askNeighborsSkyline($\mathcal{B}_{sky}$, $\mathcal{B}_{RI}$);
4: self.*cache*.storeResult($\mathcal{B}_{sky} \cap \mathcal{B}_{rec}$);
5:
6: **II. receive a skyline query ($\mathcal{B}_{sender}$)**
7: $\mathcal{B}_{RI} = \{B \in \mathcal{B}_{RI} | B$ not provided by *sender*$\}$;
8: $\mathcal{B}_{sky}$ = skyline($\mathcal{B}_{RI} \cup \mathcal{B}_{local} \cup \mathcal{B}_{sender}$);
9: askNeighborsSkyline($\mathcal{B}_{sky}$, $\mathcal{B}_{RI}$);
10: self.*cache*.storeResult($\mathcal{B}_{sky} \cap \mathcal{B}_{rec}$);
11:
12: **III. receive a skyline answer ($\mathcal{B}_{answer}$)**
13: $\mathcal{B}_{sky}$ = skyline(self.*cache*.getResult() $\cup \mathcal{B}_{answer}$);
14: **if** all queried neighbors have answered **then**
15:    **if** self is initiator **then**
16:       outputResultToTheUser($\mathcal{B}_{sky}$);;
17:    **else**
18:       sendSkylineAnswer($\mathcal{B}_{sky}$);
19:    **end if**
20: **else**
21:    self.*cache*.storeResult($\mathcal{B}_{sky}$);
22: **end if**

Part II of Algorithm 20 sketches how any peer in the system reacts upon receiving a query from one of its neighbors. There are only a few differences in comparison to part I. One of them is that an arbitrary peer has to exclude those DDS regions from consideration that describe the data of the query's sender. The second difference is that for processing the skyline on regions the peer does not only consider its local data and the information provided by its routing index but also the records received along with

the query as they might help to prune further regions and records from consideration. When the peer forwards the query to relevant neighbors, it attaches all those *records* to the query that are contained in the result of its local computation ($\mathcal{B}_{sky} \setminus \mathcal{B}_{RI}$). Thus, this set of forwarded records might contain records of local origin but also records originating from the peer the query has been received from. As intermediate result the peer stores only records into the cache that are of local origin ($\mathcal{B}_{sky} \cap \mathcal{B}_{rec}$). Those that have been received along with the query and that are still contained in the local computation of $\mathcal{B}_{sky}$ do not have to be remembered because (i) there will be no need to send them back towards the initiator as it is their origin and (ii) all answer records received from queried neighbors will have been checked for dominance by those records as they have been forwarded along with the query.

Algorithm 21 *askNeighborsSkyline($\mathcal{B}_{sky}$, $\mathcal{B}_{RI}$)*

```
1: P_ask = {n ∈ self.neighbors | ∃ a ∈ B_sky ∩ B_RI   ∧   a ∈ self.index.getRegionsFor(n)};
2: if P_ask = ∅ then
3:    if self is initiator then
4:       outputFinalResult(self.cache.getResult());
5:    else
6:       sendAnswer(self.cache.getResult());
7:    end if
8: else
9:    for all n ∈ P_ask do
10:      forwardQuery(n, B_sky \ B_RI);
11:   end for
12: end if
```

Finally, part III describes how to process answer messages. At first, the peer checks the received answer records for dominance with the intermediate result stored in the cache and obtains $\mathcal{B}_{sky}$. In case the answer messages from all queried neighbors have been received, the final result $\mathcal{B}_{sky}$ is either output to the user or sent with an answer message to the peer the query has been received from in the first place. If not all queried peers have answered yet, the intermediate result $\mathcal{B}_{sky}$ is stored into the cache.

6.3 Constraints

In the previous sections we have discussed how to process top-N and skyline queries based on local data records and DDS regions. This already reduces execution costs by reducing the number of peers that the query needs to be forwarded to. What these algorithms do not yet consider are constraints. For this purpose, we need to extend the algorithms we have presented so far. We also need to extend the plan operators that we use to represent the queries (Definition 3.3.1). For simplicity, let us assume constraints are defined as part of skyline and top-N POPs.

Let $\mathcal{A}_{rec}$ denote the set of indexed attributes, then the user-defined set of constraints $\mathcal{C}_{user}$ consists of constraints that adhere to Definition 6.3.1.

Definition 6.3.1 (Constraint)**.** *A constraint c is a triple $\langle a, low, high \rangle$, where*

- *$a \in \mathcal{A}_{rec}$ is an attribute and*
- *$low \in \mathbb{R}$ and $high \in \mathbb{R}$ define the interval $[low, high]$ that constrains attribute a.*

With $\mathcal{C}_{user}$, only data records whose attribute values lie in the defined intervals have to be considered and processed in order to answer the query. Attributes for which no constraints are given are not tested.

In order to extend the algorithms presented in the previous section to work with constraints, all we need to do is adapting the set of data records the algorithms consider as input for processing the query. So far the algorithm uses $\mathcal{B}_{local}$ computed on $\mathcal{B}_{rec}$ as input. Thus, we simply need to redefine $\mathcal{B}_{rec}$ before evaluating the query on the local data (step 2 of query processing, Figure 6.1) so that each of its elements meets the constraints. The resulting set is denoted as:

$$\mathcal{B}_{rec} := \{r \in \mathcal{B}_{rec_{old}} | \forall c \in \mathcal{C}_{user} : r[c.a] \geq c.low \wedge r[c.a] \leq c.high\} \tag{6.11}$$

The same needs to be done for $\mathcal{B}_{RI}$:

$$\mathcal{B}_{RI} := \{B \in \mathcal{B}_{RI_{old}} | \forall c \in \mathcal{C}_{user} : B.high[c.a] \geq c.low \wedge B.low[c.a] \leq c.high\} \tag{6.12}$$

Thus, $\mathcal{B}_{RI}$ now is the set of DDS regions that at least partially overlap the data space defined by the constraints, i.e., all elements have been removed that do not fulfill the constraints. Both the top-N and the skyline algorithm are working with the same data as input. If we replace $\mathcal{B}_{RI}$ and $\mathcal{B}_{rec}$ as indicated above for both algorithms, we have successfully extended them to consider constraints.

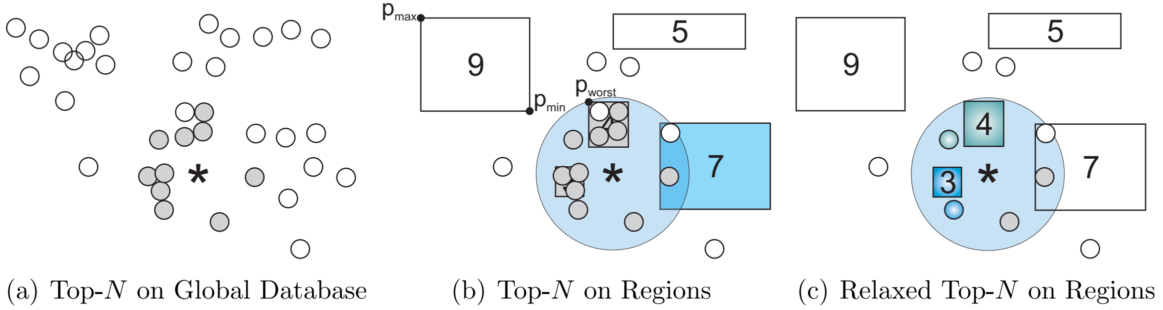
6.4 Relaxation

After having discussed how to process top-N and skyline queries with and without constraints in the previous sections, let us now consider how to reduce execution costs even further by introducing representation-based approximation. We first discuss how to extend the top-N query processing strategy in an appropriate fashion and proceed with skyline queries.

6.4.1 Top-N

Let us at first consider an example to illustrate the principle of the kind of relaxation we propose. We use the same scenario that we have already introduced in Figure 6.2. Thus, assume a top-N query $\mathfrak{T}_r^N$ asks for the top 10 elements near the asterisk (Figure 6.4). Figure 6.4(a) shows the answer to the query (grey circles) we would obtain if we computed the query on a global database that covered all the data in the network. To illustrate what data records the initiator would have considered, Figure 6.4(a) shows them as well (white circles). Figure 6.4(b) shows the result records (grey circles) that we would obtain if we used the algorithm for processing top-N queries in consideration of DDS regions. Of course, the result records are the same. However, the number of considered non-result records (white circles), which have been received at the initiator from neighboring peers, is low – only 2 in this example. Finally, Figure 6.4(c) illustrates the relaxed answer to the same query: the blue shaded record of local origin represents all records within the blue shaded region and the green shaded record of local origin represents all records within the green shaded region. As this example illustrates, the number of records that

have to be considered by the initiator is even lower than in Figure 6.4(b). The key idea in reducing execution costs by applying relaxation is that all relevant regions that can be represented by local data records do not have to be considered for forwarding the query. Thus, peers that only provide data with respect to those regions do not have to be queried and execution costs are reduced.



(a) Top-N on Global Database (b) Top-N on Regions (c) Relaxed Top-N on Regions

Figure 6.4: Relaxed Top-N Query Example

This kind of relaxation is especially useful when peers provide data clusters, i.e., most of the records that a peer provides have similar attribute values. These records can efficiently be summarized by DDSs with low approximation error. Thus, it is likely that a neighbor with similar data finds local records which can represent the data of its neighbor.

After having introduced the concept of relaxation with an example, let us now give a formal definition (Definition 6.4.1).

Definition 6.4.1 (Representative, Representing Record, Represented Region)**.** *A representative $\mathfrak{R}$ is a pair $\langle p, B\rangle$, where*

- *B is the subspace of the data space (denoted as the* represented region*) containing all records q that can be represented by $\mathfrak{R}$ and*
- *p is the* representing record *that represents B and all possible records within.*

There are two possibilities to define the represented region B:

- *B is defined by explicitly given boundaries as a subspace in the data space or*
- *B is defined by distance function $d_a : D \times D \longrightarrow \mathbb{R}$ and a maximum distance $\varepsilon \in \mathbb{R}$, i.e., B is the region containing all records q for that $d_a(q) \leq \varepsilon$ holds.*

Based on this definition we can define the relaxed result of a top-N query according to Definition 6.4.2.

Definition 6.4.2 (Relaxed Top-N Query Result)**.** *Given a set D of data records, a top-N query $\mathfrak{T}_r^N$, and a set of representatives $\mathcal{S}_{\mathfrak{R}}$, then any subset $\mathcal{B}_{topN}$ of $D \cup \mathcal{S}_{\mathfrak{R}}$ for that*

$$\forall p \in \mathfrak{T}_r^N(D)\ \exists q \in \mathcal{B}_{topN} : q = p\ \vee\ q \in \mathcal{S}_{\mathfrak{R}} \wedge q \text{ represents } p$$

holds is called a relaxed top-N query result*.*

Given these definitions, a relaxed top-N query result is a set $\mathcal{B}_{topN}$ containing either a representative for each top-N result record that would be part of the non-relaxed (exact) answer to the query or the record itself. The distance function d_a (in the example of Figure 6.4(c) we used the Euclidean Distance) describes how to determine the distance between a representing record p and a record q that is represented. Finally, ε represents the maximum distance that p and q in dependence on d_a might have. It is often possible that a record can be represented by multiple representatives. Thus, there are usually multiple result sets containing different representatives that nevertheless fulfill Definition 6.4.2 and correctly represent each record that would be contained in the exact answer to the query.

Distributed Processing of Relaxed Top-N Queries

Since the ε parameter in conjunction with the distance function d_a limits the maximum approximation the algorithm is allowed to use, both have to be specified in advance and added to the query definition. The algorithm for processing relaxed top-N queries is based on Algorithm 18 and optionally considers constraints. In order to apply relaxation, we need an additional step that tries to find representatives after having evaluated the query in consideration of the regions. In analogy to Section 6.1.2 Algorithm 22 sketches how each peer reacts upon receiving a query. It can be summarized by the following steps:

- computing the top-N query result $\mathcal{B}_{topN}$ on regions in consideration of the local data, DDS regions, the constraints, and $s_{worst_{sender}}$,
- trying to find representatives for all regions $B \in \mathcal{B}_{topN}$ using distance function d_a and the maximum approximation ε,
- forwarding a top-K query containing s_{worst} to all neighbors providing data for regions that could not be represented by representatives,
- receiving all answers and combining them with the result obtained from local computation (containing representatives) into a preliminary top-N result, and
- trying to minimize the number of representatives and forwarding the result to the peer the query has been received from (or output the result to the user).

Some of these steps are straightforward, others are not. One of the more complex steps is how to find representatives. Furthermore, a peer needs to combine the results of multiple neighbors. As these partial results might contain representatives, we have to define how to evaluate a top-N query on representatives. Finally, a peer might try to minimize the number of representatives. Thus, these are the aspects we will discuss on the following pages.

Determine Representatives

In order to reduce the number of neighbors that a query has to be forwarded to, we need to find representatives for relevant regions. According to Definition 6.4.1 a representative consists of a representing record and a represented region. To determine representing records, a peer considers all data records it stores locally. In the ideal case, all regions

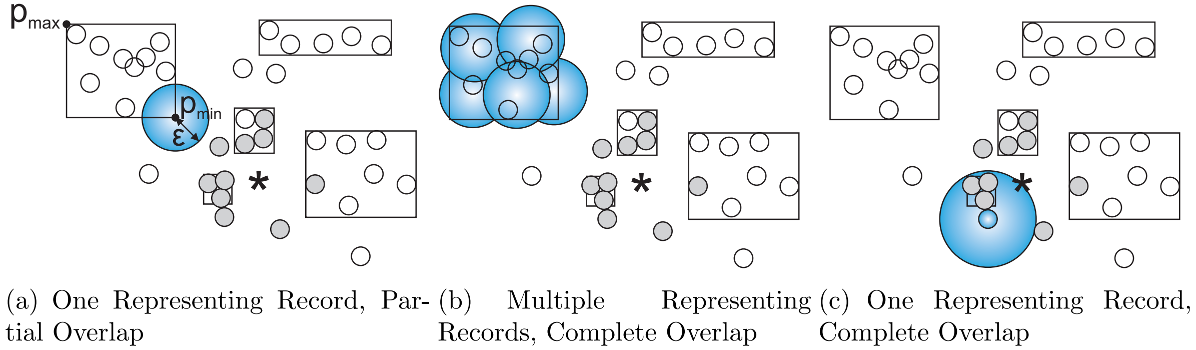
Algorithm 22 *Relaxed Top-N Query Processing*

1: **I. initiate a top-N query** ($\mathcal{B}_{local}$,$\mathcal{C}_{user}$,d_a,ε)
2: $\mathcal{B}_{RI}$ = removeEntries($\mathcal{B}_{RI}$,$\mathcal{C}_{user}$);
3: s_{worst},$\mathcal{B}_{topN}$ = topN($\mathcal{B}_{RI}$, $\mathcal{B}_{local}$);
4: $\mathcal{B}_{topN}$ = tryToReplaceRegionsWithRepresentatives($\mathcal{B}_{rec}$,d_a,ε,$\mathcal{B}_{topN}$);
5: askNeighborsTopN($\mathcal{B}_{topN}$,s_{worst},$\mathcal{B}_{RI}$);
6: self.*cache*.storeResult($\mathcal{B}_{topN} \setminus \mathcal{B}_{RI}$);
7:
8: **II. receive a top-N query** ($\mathcal{B}_{local}$,$s_{worst_{sender}}$,$\mathcal{C}_{user}$,d_a,ε)
9: $\mathcal{B}_{RI} = \{B \in \mathcal{B}_{RI} | B$ not provided by *sender*$\}$;
10: $\mathcal{B}_{RI}$ = removeEntries($\mathcal{B}_{RI}$,$s_{worst_{sender}}$);
11: $\mathcal{B}_{RI}$ = removeEntries($\mathcal{B}_{RI}$,$\mathcal{C}_{user}$);
12: $\mathcal{B}_{local}$ = removeEntries($\mathcal{B}_{local}$,$s_{worst_{sender}}$);
13: s_{worst},$\mathcal{B}_{topN}$ = topN($\mathcal{B}_{RI}$, $\mathcal{B}_{local}$);
14: $\mathcal{B}_{topN}$ = tryToReplaceRegionsWithRepresentatives($\mathcal{B}_{rec}$,d_a,ε,$\mathcal{B}_{topN}$);
15: askNeighborsTopN($\mathcal{B}_{topN}$,s_{worst},$\mathcal{B}_{RI}$);
16: self.*cache*.storeResult($\mathcal{B}_{topN} \setminus \mathcal{B}_{RI}$);
17:
18: **III. receive a top-N answer** ($\mathcal{B}_{answer}$)
19: $\mathcal{B}_{topN}$ = topN(self.*cache*.getResult() $\cup$ $\mathcal{B}_{answer}$);
20: **if** all queried neighbors have answered **then**
21:    **if** self is initiator **then**
22:       outputResultToTheUser($\mathcal{B}_{topN}$);
23:    **else**
24:       $\mathcal{B}_{topN}$ = minimizeRepresentatives($\mathcal{B}_{topN}$);
25:       sendTopNAnswer($\mathcal{B}_{topN}$);
26:    **end if**
27: **else**
28:    self.*cache*.storeResult($\mathcal{B}_{topN}$);
29: **end if**

B in $\mathcal{B}_{topN}$ can be replaced with representatives such that the query does not have to be forwarded at all.

Thus, for each region $B \in \mathcal{B}_{topN}$ the peer tries to define representatives with respect to the query definition (ranking function r, distance function d_a, and maximum distance ε). Since due to the approximation we do not know the exact attribute values of the records located within and represented by B, it is not sufficient to define only one representative for the best record p_{min} that is possibly represented by region B. Figure 6.5(a) illustrates this issue with an example. Let us assume region B (upper left corner) has been identified as relevant, we have chosen to represent it with $p_{min}(B)$, and we have found a local data record whose attribute values correspond to the ones of $p_{min}(B)$. Then, the part of B that would actually be represented by $p_{min}(B)$ is the intersection with the blue circle, whose radius is defined by d_a and ε and whose location is determined by $p_{min}(B)$ as its center. However, the part of B that is represented does not contain any data record in the real data set. Thus, choosing p_{min} as representing record is incorrect. However, if we increased ε so that the circle defined by it around $p_{min}(B)$ completely contained region B, representing B with a representative based on p_{min} would be correct.

However, it is still possible to represent the region although we do not increase ε. Figure 6.5(b) illustrates the ideal solution. If we allow a representative to consist of multiple representing records and the regions defined by ε, d_a, and the records completely enclose region B, then each record that is possibly contained in B is correctly represented. As it is a hard problem to find an optimal set of representing records, we favor a heuristic that defines a representative based on only one data record whose ε-region completely encloses the represented region B (Figure 6.5(c)).



(a) One Representing Record, Partial Overlap (b) Multiple Representing Records, Complete Overlap (c) One Representing Record, Complete Overlap

Figure 6.5: Finding Representatives for Regions

Figure 6.5(c) also illustrates that the maximum region (big circle highlighted in blue) that could be represented by the representing record (small circle highlighted in blue) is much larger than the actually represented region (rectangle highlighted in blue). Thus, informing the user only about the representing record, d_a, and ε (that altogether determine the maximum region) makes the result more fuzzy than necessary. A better description of the represented data can be given by explicitly specifying the represented region, which has to be completely contained in the maximum region defined by the representing record, d_a, and ε. For this purpose, we can simply use the boundaries of the DDS region that we want to represent – applying Definition 6.4.1's second option to define the represented region.

Top-N Computation on Representatives

Having received all the answer messages from all queried neighbors, a peer needs to evaluate the top-N query on the union of the received partial results and its local intermediate result. As both result sets might contain representatives, we need to define how to evaluate a top-N query on the union of data records and representatives. As a representative is defined as a pair $\langle p, B \rangle$, we can use B to determine a lowest and a highest score (s_{min} and s_{max}) for a representative. In doing so, we can use the same algorithms as introduced above with only negligible adaptations to consider p and B.

Minimizing the Number of Representatives

In order to reduce network traffic, a peer tries to minimize the number of representatives before forwarding its own answer to a neighbor. For this purpose, we split up the representatives. We obtain two lists: the first one $\mathcal{F}$ consists of all representing records and the second one $\mathcal{G}$ consists of all represented regions. The goal is to find a minimal subset of $\mathcal{F}$ that correctly represents all regions in $\mathcal{G}$. As this is the NP-complete problem SET-COVER, we use the heuristic sketched in Algorithm 23 to efficiently compute a solution to this problem.

At first, the set of representing records $\mathcal{F}$ is sorted in descending order by the number of regions $B \in \mathcal{G}$ that each $p \in \mathcal{F}$ could represent with respect to the distance function d_a and the maximum approximation ε. For each region $B \in \mathcal{G}$ we first try to choose a representative $\langle p', B' \rangle$ that was already found in previous iterations and that with respect

to d_a and ε is able to represent B as well. In case we can find such a representative, we adapt its represented region so that both B and B' are represented by p'. If we cannot find such a representative, we define a new representative (p, B) and choose an appropriate $p \in \mathcal{F}$ by traversing $\mathcal{F}$ from top to bottom. As $\mathcal{F}$ has been sorted according to the number of regions a record might represent, the records that can represent the most regions are considered first and thus favored. After having found representatives for all regions $B \in \mathcal{G}$, the number of representatives has hopefully been reduced.

Algorithm 23 *minimizeRepresentatives(representatives)*

```
1: F = extractRecords(representatives);
2: G = extractRegions(representatives);
3: sort(F); // in descending order by the number of regions each p ∈ F could represent
4: chosen = ∅; // set of pairs (p, B) so that p represents B
5: for all B ∈ G do
6:    if ∃ (p′, B′) ∈ chosen, so that p′ represents B ∪ B′ then
7:       (p′,B′) = (p′,B ∪ B′);
8:    else
9:       find p ∈ F, so that p represents B;
10:      chosen.add((p, B));
11:   end if
12: end for
13: return chosen;
```

6.4.2 Skylines

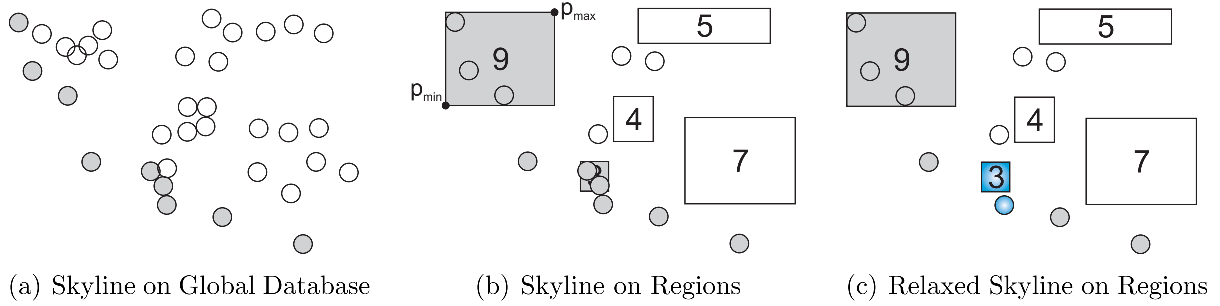
Before we give a formal definition of relaxed skylines, let us first consider an example based on Figure 6.3. Assuming the skyline query $\mathfrak{S}_{\mathcal{R}}$ is defined on two ranking functions that are both annotated with MIN, then Figure 6.6(a) shows the answer to the query (grey highlighted circles) that would be obtained if we had a global database consisting of all data records provided by all peers altogether. Figure 6.6(b) shows the result that would be obtained in consideration of DDS regions – result records and relevant regions are again highlighted in grey. Furthermore, the figure shows all further non-result records the query initiator would have received from neighboring peers and considered for local evaluation. In comparison to Figure 6.6(a) we see that the number of records considered by the initiator is reduced due to the consideration of DDS regions. Finally, Figure 6.6(c) shows how the result in consideration of regions and representatives might look like. As already described in the context of top-N queries, the goal is to replace relevant regions with representatives so that the respective neighbors do not have to be queried. In this example, we have one representative (highlighted in blue). As a consequence, the number of records and regions that have to be considered is reduced in comparison to Figure 6.6(b).

Definition 6.4.3 (Relaxed Skyline Query Result)**.** *Given a set D of data records, a skyline query $\mathfrak{S}_{\mathcal{R}}$, and a set of representatives $\mathcal{S}_{\mathfrak{R}}$, then any subset $\mathcal{B}_{sky}$ of $D \cup \mathcal{S}_{\mathfrak{R}}$ for that*

$$\forall p \in \mathfrak{S}_{\mathcal{R}}(D)\ \exists q \in \mathcal{B}_{sky} : q = p\ \vee\ q \in \mathcal{S}_{\mathfrak{R}} \wedge q \textit{ represents } p$$

holds is called a relaxed skyline query result*.*

Given Definition 6.4.3, a relaxed skyline query result is a set $\mathcal{B}_{sky} \subseteq D \cup \mathcal{S}_{\mathfrak{R}}$ that either contains a representative for each record $p \in \mathfrak{S}_{\mathcal{R}}(D)$ that would be part of the



(a) Skyline on Global Database (b) Skyline on Regions (c) Relaxed Skyline on Regions

Figure 6.6: Relaxed Skyline Query Example

skyline evaluated on all data records or p itself. The definition of representatives follows Definition 6.4.1. Just as for relaxed top-N queries, there are often multiple records that could represent a region and the same record can represent multiple regions. This is an advantage in comparison to classic skyline computation as especially for the anticorrelated data set (as shown in [19]) the result set consists of a very high number of records. By introducing relaxed skylines the result size can considerably be reduced and the user is still provided with the "big picture".

The general tendency is obvious, the higher the user-defined maximum relaxation ε, the bigger are the regions that can be represented by representatives and thus the lower are query execution costs. In general, the loss in detail due to the application of approximation can be neglected. A user most likely issues a skyline query to find out about his/her options without having to define which ranking function $r \in R$ is more important to him/her in advance. Thus, the relaxed skyline result might help him/her to come to a decision. Afterwards, in case the user needs more detailed information about a specific represented region, he/she can still issue a more specific query. The overview, however, was computed efficiently and presented already after a short time in comparison to the exact computation.

Distributed Processing of Relaxed Skylines

The algorithm for processing relaxed skyline queries (sketched in Algorithm 24) is based on the baseline algorithm of Algorithm 20 and optionally considers constraints. Just as for relaxed top-N queries we assume that the distance function d_a as well as the maximum approximation ε are part of the query's definition. The main difference to the baseline algorithm is the introduction of an additional step, which tries to find representatives for the regions identified as being relevant to the query. The algorithm can be summarized by the following steps that are executed at each peer:

- computing the skyline on regions $\mathcal{B}_{sky}$ on the union of the peer's local data, DDS regions, and the records received along with the query,
- trying to find representatives for all regions $B \in \mathcal{B}_{sky}$ using distance function d_a, the maximum approximation ε, all local records, and the records received along with the query,

- forwarding the query to all peers providing data in regions that could not be represented by representatives, all records $p \in \mathcal{B}_{sky}$ (no regions and representatives) are forwarded along with the query,
- receiving the answers from all queried neighbors and determining the skyline on the union of the partial results received from the neighbors and the local result – the latter containing representatives for the data of neighbors that have not been queried, and
- trying to minimize the number of representatives that are part of the skyline computed in the previous step and forwarding the resulting skyline to the query's sender or outputting it to the user.

In the context of processing top-N queries we have already described how to determine representatives and how to combine them. We can use the same algorithms without further adaptations in the context of skyline queries. However, what we still need to discuss is how to compute a skyline on representatives.

Algorithm 24 *Relaxed Skyline Query Processing*

1: **I. initiate a skyline query** ($\mathcal{B}_{local}$,$\mathcal{C}_{user}$,d_a,ε)
2: $\mathcal{B}_{RI}$ = removeEntries($\mathcal{B}_{RI}$,$\mathcal{C}_{user}$);
3: $\mathcal{B}_{sky}$ = skyline($\mathcal{B}_{RI} \cup \mathcal{B}_{local}$);
4: $\mathcal{B}_{sky}$ = tryToReplaceRegionsWithRepresentatives($\mathcal{B}_{rec}$,d_a,ε,$\mathcal{B}_{sky}$);
5: askNeighborsSkyline($\mathcal{B}_{sky}$, $\mathcal{B}_{RI}$);
6: self.*cache*.storeResult($\mathcal{B}_{sky} \setminus \mathcal{B}_{RI}$);
7:
8: **II. receive a skyline query** ($\mathcal{B}_{local}$,$\mathcal{B}_{sender}$,$\mathcal{C}_{user}$,d_a,ε)
9: $\mathcal{B}_{RI} = \{B \in \mathcal{B}_{RI} | B$ not provided by *sender*$\}$;
10: $\mathcal{B}_{RI}$ = removeEntries($\mathcal{B}_{RI}$,$\mathcal{C}_{user}$);
11: $\mathcal{B}_{sky}$ = skyline($\mathcal{B}_{RI} \cup \mathcal{B}_{local} \cup \mathcal{B}_{sender}$);
12: $\mathcal{B}_{sky}$ = tryToReplaceRegionsWithRepresentatives($\mathcal{B}_{rec} \cup \mathcal{B}_{sender}$,$d_a$,$\varepsilon$,$\mathcal{B}_{sky}$);
13: askNeighborsSkyline($\mathcal{B}_{sky}$, $\mathcal{B}_{RI}$);
14: self.*cache*.storeResult($\mathcal{B}_{sky} \setminus \mathcal{B}_{RI}$);
15:
16: **III. receive a skyline answer** ($\mathcal{B}_{answer}$)
17: $\mathcal{B}_{sky}$ = skyline(self.*cache*.getResult() $\cup$ $\mathcal{B}_{answer}$);
18: **if** all queried neighbors have answered **then**
19:    **if** self is initiator **then**
20:       outputResultToTheUser($\mathcal{B}_{sky}$);;
21:    **else**
22:       $\mathcal{B}_{sky}$ = minimizeRepresentatives($\mathcal{B}_{sky}$);
23:       sendSkylineAnswer($\mathcal{B}_{sky}$);
24:    **end if**
25: **else**
26:    self.*cache*.storeResult($\mathcal{B}_{sky}$);
27: **end if**

Skyline Computation on Representatives

So far we use $\prec_{\mathcal{R}_{Reg}}$, as defined in Equations 6.5 through 6.8, to compute the skyline on data records and regions. In order to compute a skyline in consideration of representatives, we extend the dominance relation once more to $\prec_{\mathcal{R}_{Rep}}$. As we do not know exactly what records are represented (i.e., we do not know their attribute values) but only the region that is represented, we can again use p_{min} and p_{max} to decide on dominance. Given

two representatives $R_1 = (p_1, B_1)$ and $R_2 = (p_2, B_2)$, then R_1 dominates R_2 if $p_{max}(B_1)$ dominates $p_{min}(B_2)$, thus:

$$p_{max}(B_1) \prec_{\mathcal{R}} p_{min}(B_2) \Rightarrow B_1 \prec_{\mathcal{R}_{Reg}} B_2 \Rightarrow R_1 \prec_{\mathcal{R}_{Rep}} R_2 \quad (6.13)$$

Furthermore R_1 dominates R_2 if the representing record of R_1 dominates the region represented by R_2, thus:

$$p_1 \prec_{\mathcal{R}_{Reg}} B_2 \Rightarrow R_1 \prec_{\mathcal{R}_{Rep}} R_2 \quad (6.14)$$

Consequently, in order to determine if a representative dominates another representative, we simply need to evaluate the conditions defined in Equations 6.13 and 6.14. For combinations of representatives and regions/records, we simply extract the information about the represented region and determine the dominance relationship according to Equations 6.5 through 6.8.

6.5 Evaluation

In this chapter we have introduced techniques and algorithms for efficiently processing rank-aware queries based on DDSs. In order to evaluate their performance, we implemented all these algorithms in SmurfPDMS (Chapter 7). In order to eliminate interfering side-effects of the rewriting component, we performed all tests in a network with peers using the same schema. We focused on four different setups, each is based on the same cycle-free topology of 100 peers but uses different data sets. We also performed the same tests in bigger and smaller networks but, as expected, found the same tendencies. Thus, in the following we focus on our results for the network of 100 peers with each peer providing 50 four-dimensional data records and having 1 to 3 neighbors. All dimensions/attribute values are restricted to the interval $[0, 1000]$ and indexed by DDSs with four-dimensional base structures. The four setups that we consider are:

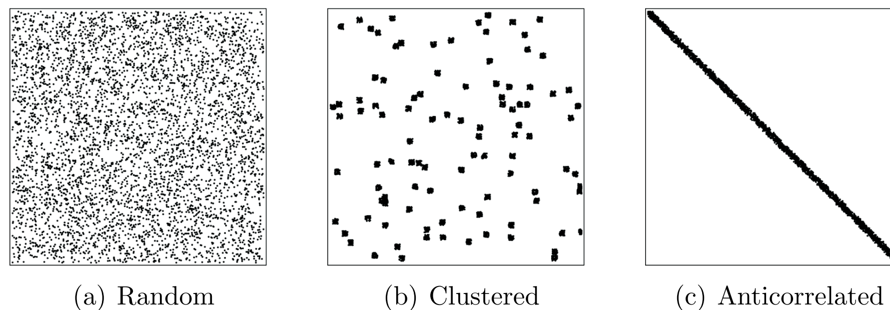
1. *Random Data, Random Distribution*: For each peer 50 data records are created. The attribute values of each data record and for all dimensions are chosen randomly but restricted to the interval $[0, 1000]$. Each attribute value is determined independently from the others. Figure 6.7(a) depicts the data that all 100 peers provide altogether.

2. *Clustered Data, Random Distribution*: 100 cluster centers are chosen in the style of random data. 5000 data records are created (50 per cluster) by offsetting each of the cluster center's four attribute values by adding a random value of the interval $[-10, 10]$. The resulting 5000 records are distributed randomly among all 100 peers so that each peer is assigned a set of 50 records, which are likely to originate from different clusters. Figure 6.7(b) shows the set of 5000 records.

3. *Clustered Data, Clustered Distribution*: The same data set is used as in the previous scenario but distributed differently among the peers. In this setup each peer is assigned one of the 100 clusters so that each peer provides all 50 records of one cluster.

4. *Anticorrelated Data, Random Distribution*: 5000 records are created by randomly choosing a point for each record on the straight line through $p_1(1000, 0, 0, 1000)$ and $p_2(0, 1000, 1000, 0)$. This randomly chosen point is offset by a random value of $[-10, 10]$ for each dimension. Each peer is assigned a set of 50 records. Figure 6.7(c) shows the corresponding data that all peers provide altogether.

The DDSs used in our tests were defined on all four attributes/dimensions. In our tests we used QTree-based DDSs as well as multidimensional equi-width histogram-based DDSs – in the following referred to as *QDDSs* and *HDDSs*. The parameters for QDDSs were $b_{max} = 50$ and $f_{max} = 4$. HDDSs were allowed five buckets per dimension, i.e., 625 in total for each neighbor. In all our tests we varied the relaxation parameter ε from 0 to 1100 and applied the Euclidean Distance as distance function d_a. To enable comparability, in each test the same peer issued the same query. In order to make use of our multidimensional index structures, the top-N ($N = 500$) test query is defined on two attributes by the following ranking function: $attribute1 + attribute2$. Likewise, the skyline test query is defined as $MIN(attribute1)$ and $MIN(attribute2)$.



(a) Random (b) Clustered (c) Anticorrelated

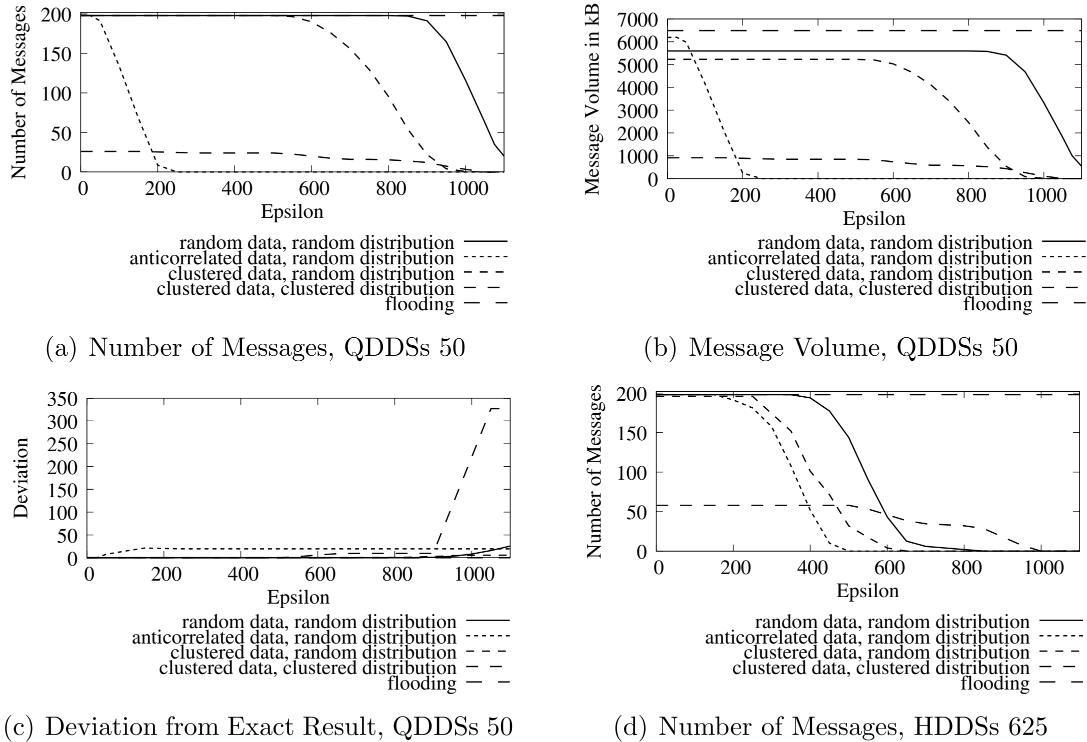
Figure 6.7: Two-Dimensional Projection of the Four-Dimensional Test Data Sets

6.5.1 Top-N Queries

To evaluate our techniques for top-N query processing, let us first discuss Figures 6.8(a), 6.8(b), and 6.8(c) in that we used QDDSs with $b_{max} = 50$. These three figures show how (a) the number of messages (answer and query), (b) the message volume, and (c) the deviation from the exact answer to the query change with increasing values for the approximation parameter ε. As the general reduction of the number of messages for all four scenarios in Figure 6.8(a) indicates, in accordance with our intention, the application of relaxation reduces query execution costs, i.e., the higher ε the greater is the cost reduction in terms of the number of messages. Since the message volume is closely related to the number of messages, it is reduced as well with increasing ε – Figure 6.8(b).

Apart from this general tendency, these results permit drawing conclusions concerning the amount of cost reduction that originates from the mere use of DDSs: the test run with ε set to 0 corresponds to the exact algorithm, i.e., DDSs are considered but relaxation is not applied. In case of scenario 3 (clustered data in a clustered distribution), by the mere use of DDSs the number of messages necessary to answer the query is reduced to less than 15% in comparison to the alternative of flooding. The reason for this effect is that this is the best scenario for any DDS since each peer provides data in only one cluster

so that these clusters can be represented easily with low approximation error. Using this information enables the strategy to prune many peers from consideration, which results in the reduction in the number of messages and also in the reduction of data volume. Thus, cost reduction depends not only on the applied degree of approximation but also on the data distribution and the quality of DDS summarization.



(a) Number of Messages, QDDSs 50

(b) Message Volume, QDDSs 50

(c) Deviation from Exact Result, QDDSs 50

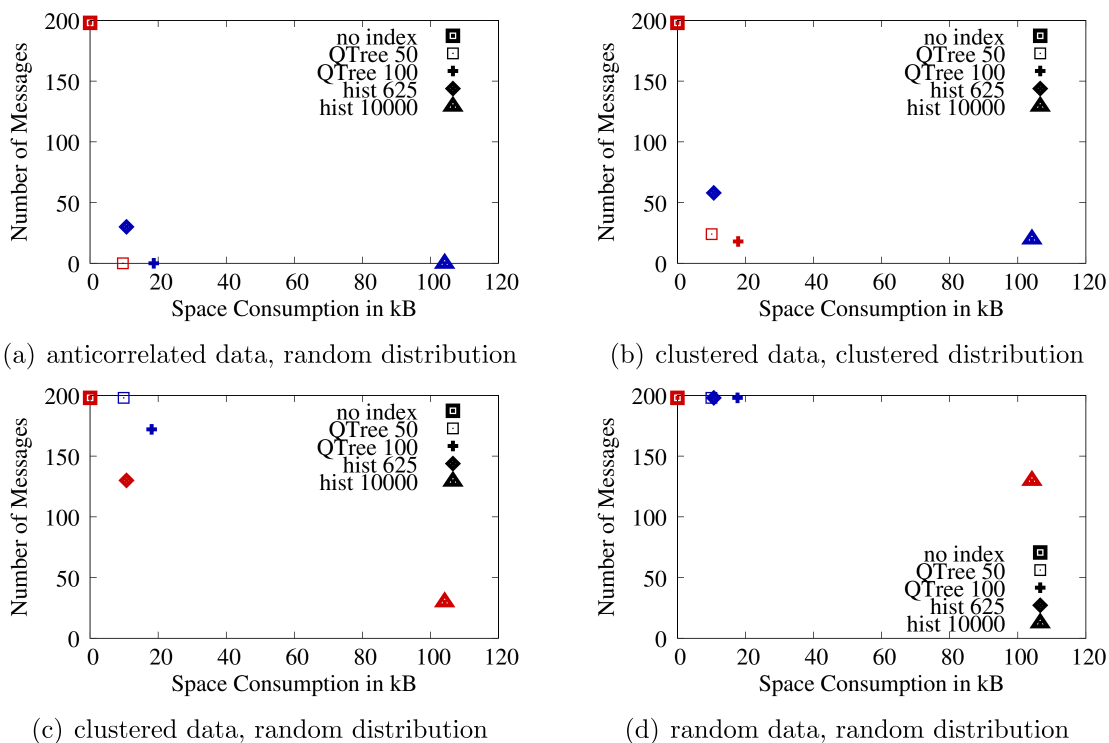
(d) Number of Messages, HDDSs 625

Figure 6.8: Distributed Processing of Relaxed Top-N Queries

Another question we want to answer is: how does the application of approximation influence result quality? Figure 6.8(c) answers this question. It shows the average deviation (Euclidean distance) of each record of the exact result ($\epsilon = 0$) to its corresponding representing record that has been output to the user. Note that this does not mean the algorithm has made an error. It is only the consequence of the allowed relaxation that has been seized. In comparison to Figures 6.8(a) and 6.8(b) we see that the reduction of execution costs achieved by applying approximation is directly connected to the increase in the average deviation to the exact result.

In order to compare QDDSs and HDDSs, we performed the same tests with HDDSs as well – Figure 6.8(d) shows our results with respect to the number of messages. The QDDSs that we used in the previous tests were allowed 50 buckets to summarize the data of all neighbors altogether whereas the HDDSs in these tests were allowed 625 buckets for each neighbor. However, we still consider this to be a fair comparison as in our implementation and in our network the overall consumption of memory space that both (QDDSs 50 and HDDSs 625) require is about the same. By comparing Figures 6.8(a) and 6.8(d) we see that no matter what DDS variant we use the general tendency of message reduction with increasing ε remains the same. QDDSs are the best choice for

several setups. But there are some scenarios for that HDDSs are the better choice: scenarios 1 (random data in a random distribution) and 2 (clustered data in a random distribution). The reason for this effect is that for random data there are no clusters that can be summarized easily by QDDSs. Clustered data distributed in a random fashion among all peers is likewise difficult to summarize by QDDSs – especially if we consider that each peer might provide data of 50 different clusters and a peer is only allowed to use 50 buckets to describe the data of all neighbors altogether. Thus, buckets have to be very big in order to cover the whole data space and a high value for ε is necessary to replace relevant regions with representatives. As HDDSs were allowed much more buckets, despite the firm grid of buckets that is not adapted no matter what the data distribution looks like, the buckets' regions are much smaller and hence can be replaced by representatives with lower values for ε. Increasing the number of QDDS buckets also counteracts this problem, as illustrated in Figure 6.9.



(a) anticorrelated data, random distribution

(b) clustered data, clustered distribution

(c) clustered data, random distribution

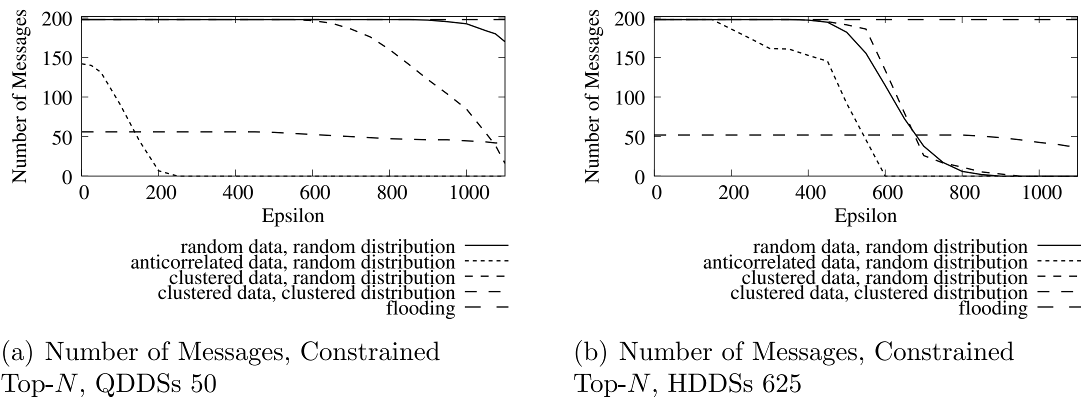
(d) random data, random distribution

Figure 6.9: Cost-Benefit-Analysis for Top-N Query Processing in Conjunction with Different Types of DDSs. $\varepsilon = 400$, *skyline points red, non-skyline points blue*

Figure 6.9 shows the dependency of execution costs (i.e., the number of messages) on the memory space required to manage DDSs. It shows some examples for QDDSs and HDDSs in our four network setups – note that the memory space strongly depends on the implementation and may therefore differ between implementations. But given our two implementations, the question we need to answer is: what is the best choice? As all too often, it depends. . .

Let us regard this as an optimization problem. We want to minimize two criteria: memory space and execution costs. And we want to find good combinations with acceptable values for both criteria. Since we cannot yet specify any reasonable boundaries for

these two criteria nor importance rankings, we have a situation where the computation of a skyline might help: the red points in Figure 6.9 represent the skylines that reveal what DDSs represent "good combinations" for both criteria using an ε of 400 in conjunction with QDDSs ($b_{max} = 50$ and $b_{max} = 100$), HDDSs (625 buckets and 10000 buckets), and a flooding approach. Of course, the point representing the absence of any index is always in the skyline because it requires no memory space at all, i.e., representing a minimal value for one of the two dimensions. In Figures 6.9(a) and 6.9(b) QDDSs clearly dominate HDDSs. However, as we have already concluded by means of Figures 6.8(a) and 6.8(d), HDDSs are the better choice for other scenarios – Figures 6.9(c) and 6.9(d). For random data (Figure 6.9(d)) we have to invest much memory space to obtain only a small reduction in execution costs.



(a) Number of Messages, Constrained Top-N, QDDSs 50

(b) Number of Messages, Constrained Top-N, HDDSs 625

Figure 6.10: Distributed Processing of Relaxed Constrained Top-N Queries

Finally, let us point out what happens if we additionally apply constraints. The constraints we applied restrict all attribute values to the interval $[500, 1000]$. One might expect that reducing the query space to a quarter might reduce execution costs since there is less relevant data in the network and thus less peers providing relevant data. However, our results illustrated in Figures 6.10(a) and 6.10(b) teach us otherwise. In comparison to full space queries, execution costs slightly increase with the application of constraints or stay more or less the same for three out of four test scenarios and for both kinds of DDSs. The reason is that for these three scenarios the set of records representing the answer to the query in the constrained space is different from the set of records representing the answer to the full space query. In fact, it is possible that they share not even a single record. Thus, many regions that have not been relevant before, suddenly become relevant and peers are queried that would not be queried for the full space query. Furthermore, as the relevant data space is reduced, the set of local records that could be used to define representatives decreases so that the chance of finding representatives is lower.

This is different for the anticorrelated data setup. In this case by defining the constraints as explained above, the network no longer provides N records that fulfill the constraints. Thus, the amount of relevant data is reduced. As QDDSs offer a very good summarization of anticorrelated data, the application of QDDSs leads to a cost reduction for constrained queries. This is supported by the fact that even without applying approximation ($\varepsilon = 0$), query execution costs are reduced – Figure 6.10(a). However,

when applying HDDSs we can hardly notice this effect. In contrast, we can even notice a slight increase in execution costs because of a contrary effect. There are only a few records and regions left that are relevant to answer the query. In conjunction with the worse approximation quality of HDDSs, i.e., regions tend to be larger in comparison to QDDSs, it is harder for a peer to define representatives based on its local data for relevant regions that neighboring peers provide data for.

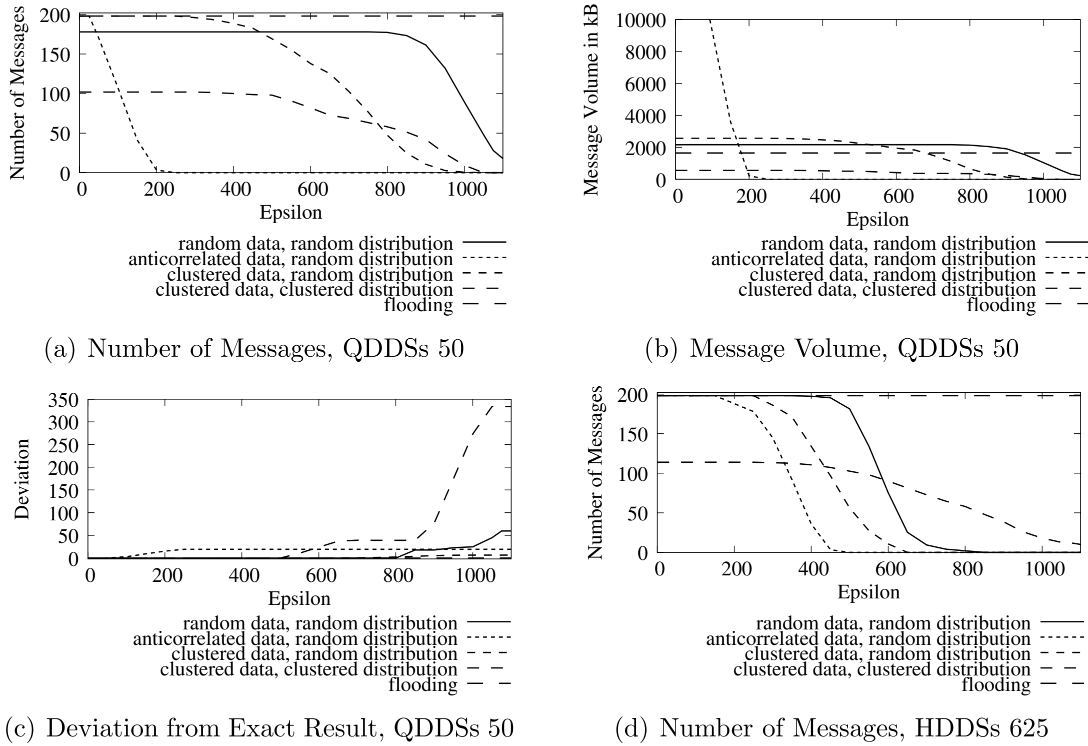
6.5.2 Skyline Queries

With respect to skyline queries we basically found the same tendencies that we have already found for top-N queries:

- applying DDSs alone without relaxation already reduces execution costs – Figure 6.11(a),
- for all four scenarios the number of messages and the corresponding data volume decrease for higher values of ε – Figures 6.11(a) and 6.11(b),
- while query execution costs decrease with increasing ε, the deviation to the exact answer increases – Figure 6.11(c), and
- all these tendencies are the same for QDDSs and HDDSs – Figures 6.11(a) and 6.11(d).

We also examined the influence of the data distribution on performance. The anticorrelated data setup in particular is expected to cause high execution costs. However, especially for this kind of data our techniques show the greatest increase in performance, although of all setups this is usually the worst case scenario for any skyline algorithm because of the potentially high number of result records [19]. As Figures 6.11(a) and 6.11(d) show, for this setup the number of messages is already low even for small values of ε in comparison to the other scenarios. This is because even a small ε allows the algorithm to represent data provided by neighboring peers. Thus, both the number of messages and the data volume are reduced. With respect to the other setups Figures 6.11(a) and 6.11(d) show the same tendencies as for top-N queries: HDDSs again allow for a better performance for higher ε in the cases of the "clustered data, random distribution" and "random data, random distribution" scenarios because QDDSs have less buckets at their disposal to summarize the data.

Figures 6.12(a) and 6.12(b) answer the question what happens when we additionally introduce constraints. In these tests we again applied constraints restricting relevant attribute values to the interval $[500, 1000]$. As these results again show, applying constraints is not always an appropriate means to reduce execution costs. On the contrary, for some scenarios query execution costs are increased. The reason is that the skyline of a constrained data space consists of different records than the full space query and thus peers that have not been relevant before become relevant and have to be considered. Furthermore, with less records being relevant it is harder to define representatives based on local data records for relevant data (regions) of neighboring peers. However, for the anticorrelated data set we again benefit from the fact that the size of the result set is smaller in comparison to the full space query. In contrast to top-N queries we also note this effect for the clustered data set in the clustered distribution: the size of the result set decreases and as a consequence of the distribution also the number of relevant peers.



(a) Number of Messages, QDDSs 50

(b) Message Volume, QDDSs 50

(c) Deviation from Exact Result, QDDSs 50

(d) Number of Messages, HDDSs 625

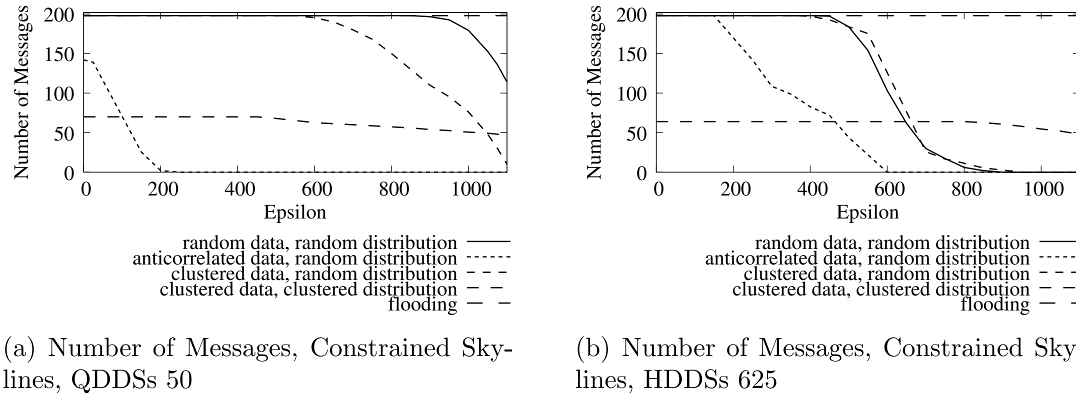
Figure 6.11: Distributed Processing of Relaxed Skylines

Finally, Figure 6.13 helps us identify what DDS structures provide a good combination of space consumption and cost reduction. With respect to skyline queries QDDSs are the best choice for the anticorrelated data, random data, and the "clustered data, clustered distribution" scenario. For a setup that corresponds to the clustered data in a random distribution scenario, with respect to skyline queries QDDSs as well as HDDSs might be preferred depending on the available space because all DDSs we tested represent good combinations of both criteria (query execution costs and space consumption).

6.6 Conclusion

In this chapter, we have introduced efficient strategies for processing rank-aware queries (top-N and skyline) in distributed environments (PDMSs as well as unstructured P2P systems in general). These strategies exploit the information provided by DDSs and may additionally consider relaxation and constraints. Our evaluation results show that there are various aspects influencing query execution costs. One of the main aspects is the data distribution. It primarily determines the quality of DDSs, i.e., in general clustered data can be summarized more efficiently than random data. Of course, the chosen DDS variant (e.g., QDDSs or HDDSs) also influences the quality of the summarization by implementation specific characteristics. Furthermore, the data distribution also influences the number of result records and thus also the amount of relevant data (and peers) that needs to be considered in order to answer a query correctly.

Another aspect influencing execution costs is the application of relaxation. Our eval-



(a) Number of Messages, Constrained Skylines, QDDSs 50

(b) Number of Messages, Constrained Skylines, HDDSs 625

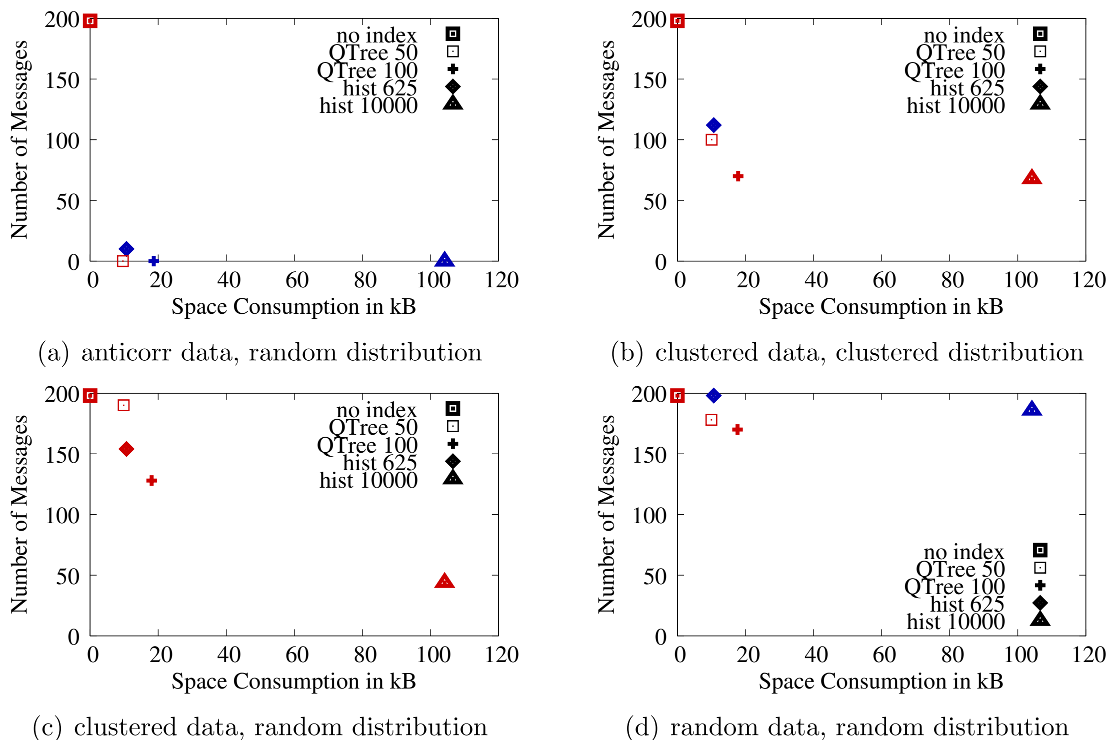
Figure 6.12: Distributed Processing of Relaxed Constrained Skylines

uation has shown, as it was our intention, that relaxation may help to reduce costs in terms of the number of sent messages and data volume. This is especially true for the anticorrelated data set in conjunction with skyline queries although most algorithms, also those for centralized systems, in general perform poorly for this kind of data. Hence, considering this technique for centralized systems could be an interesting aspect of future work.

The algorithms proposed in this chapter optionally consider constraints. With respect to rank-aware queries, the application of constraints does not necessarily mean the query load is reduced. The reason is that due to the restriction the result most likely contains totally different records in comparison to the non-restricted case. Thus, in conjunction with rank-aware queries, constraints cannot be considered an appropriate means to reduce execution costs.

Although we have only presented our results with respect to the basic query shipping approach (Section 2.5.1), i.e., each peer waits for the answers from all queried neighbors before it sends an own answer, the tendencies we found also hold for incremental strategies (IMS Section 2.5.3), i.e., each peer sends answer messages as soon as it identifies new result records, for example upon having received an answer message from one of its neighbors. In future work we might consider optimizing both strategies, e.g., by representing regions not only with single records but with a set of records as illustrated in Figure 6.5(b) or by using a query cache to reuse the result sets of previous queries and issue only compensation queries to retrieve the data that cannot be obtained from the cache.

The strategies we have proposed in this chapter forward additional information along with the query in order to help the receiving peer prune records and peers from consideration (Sections 6.1.2 and 6.2.2). So far this additional information is determined only based on data records that are guaranteed to be available, i.e., local data records of the peer forwarding the query and local data records of peers on the path to the initiator that the query has been propagated on. Extended variants of our strategies could consider not only guaranteed data but also data a peer hopes to receive from neighboring peers. In doing so, a peer acts on the assumption that it will receive the data represented by the routing indexes from its neighbors. However, peers might have left the network and the



(a) anticorr data, random distribution
(b) clustered data, clustered distribution
(c) clustered data, random distribution
(d) random data, random distribution

Figure 6.13: Cost Benefit Analysis for Skyline Query Processing in Conjunction with Different Types of DDSs. $\varepsilon = 400$, *skyline points red, non-skyline points blue*

routing indexes might not have been updated yet. If the information added to forwarded queries is based on such incorrect data, it is possible that the obtained result records do not meet the expectations. In that case, a peer needs to issue compensation queries in order to obtain the missing result records. As issuing such compensation queries is expensive, the strategies proposed in this chapter consider only guaranteed data to refine the forwarded query.

Chapter 7

SmurfPDMS

In this dissertation we have proposed techniques that extend PDMS functionality to support rank-aware query operators and approximation in conjunction with XML data. In order to evaluate our approaches, we needed to implement these techniques. As we have already pointed out in Section 2.7, existing PDMS implementations are hard to extend to support our techniques and strategies. Furthermore, most existing simulators such as ns-2 [148] are too low-level to simulate PDMSs. Thus, we decided to create a new environment that supports all the techniques we have discussed in the previous chapters.

The main requirements an appropriate environment has to fulfill are the support of XML data, heterogeneity, mappings, query rewriting algorithms, routing indexes and corresponding maintenance strategies, and query processing strategies considering rank-aware query operators and approximation/relaxation. Furthermore, as our ambition was to create an environment to help us evaluate our techniques, the resulting environment should be extensible in several ways so that in the future we can add additional strategies and compare them to others. For evaluation, the environment has to be able to use different network topologies, data distributions, and query loads. In this sense, it should be able to simulate arbitrary large-scale networks. In order to evaluate the influence of individual parameters, the environment should be able to reproduce exactly the same initial configurations of a PDMS for multiple test runs (repeatability). On a higher level, it should be platform independent and provide tools to support the user in creating initial setups and conducting experiments. Finally, a graphical user interface could visualize the simulated PDMS.

In this chapter, we present SmurfPDMS (SiMUlating enviRonment For PDMSs)[1] [88, 91,105], a simulator that fulfills all these requirements and incorporates all the techniques we have proposed in this dissertation. We decided to implement a simulator instead of a real system as in a simulator we can control external influences on the system and better measure the performance of our strategies. In this chapter, we first present the system architecture of SmurfPDMS and proceed with discussing its main components. Afterwards, we illustrate the workflow of a simulation and then go into details on some implementational issues.

[1] Only recently, we have learned that there is another project at UC Berkeley named SMURF [102], which is an approach for RFID data cleaning. The similarity in the projects' names is purely coincidental.

7.1 System Architecture

For simulating large networks (especially with a lot of data provided by the peers) the capacities of one computer might not be enough. Thus, we decided to optionally combine the resources of multiple computers, e.g., a number of PlanetLab [160] nodes. Each computer runs an instance of SmurfPDMS and simulates a partition of the *simulated network*. In this context, we have to distinguish between a (simulated) peer and a computer, which might possibly run multiple SmurfPDMS instances. When multiple instances are involved in a simulation, one of them takes on a special role: it serves as the *simulation coordinator*, which synchronizes all other instances (*participants*). Note that this coordinator is only used to run a simulation of a simulated PDMS on multiple computers and does in no way correspond to a mediator in data integration systems. However, using the simplest configuration, one computer simulates a whole network of peers and their interactions. An important task of the coordinator while running a simulation is synchronizing all participants. For this purpose, we introduced discrete time steps and a central simulation clock at the coordinator. The coordinator synchronizes all participants so that they are always in the same simulation step. In each time step at most one message is processed for each simulated peer, e.g., processing a query and possibly forwarding it to neighboring peers. In order to simulate processing time and communication delays, messages are assigned arrival times indicating the earliest time step in the future at which the message can be processed.

This kind of synchronization enables the elimination of all random delays that might occur when transferring data from one computer to another. Instead, we can "control" communication delays between peers and processing time at the peers by simulating this behavior. Furthermore, the application of discrete time steps enables repeatability, e.g., queries can be issued at exactly the same time and with exactly the same communication delays and processing times. Besides, the influence of communication delays can even be completely switched off.

The requirements stated above led to the system architecture illustrated in Figure 7.1: SmurfPDMS consists of three independent layers. The bottom layer is the *network layer*, which enables communication between SmurfPDMS instances. The middle layer (*simulation layer*) is responsible for running a simulation. Finally, the top layer (*GUI layer*) provides a graphical user interface that enables configuration, monitoring, and interaction with the system. We designed SmurfPDMS with the intention to make it extensible with respect not only to the strategies we have proposed but also to the layers. Thus, we can easily exchange the communication protocol of the network layer – we have already done so by replacing JXTA [106, 149] with TCP/IP communication. Furthermore, the simulation layer is completely independent from the GUI layer such that we can even run simulations without the graphical component. To ensure platform independence we chose Java as programming language.

In the remainder of this section, we discuss some more details on the three layers and their main components shown in Figure 7.1. Our current implementation consists of about 700 classes (about 120, 000 lines of code) organized in multiple packages. For the sake of brevity, we discuss only the most important classes – leaving out details on aspects such as exception/error handling and thread implementation.

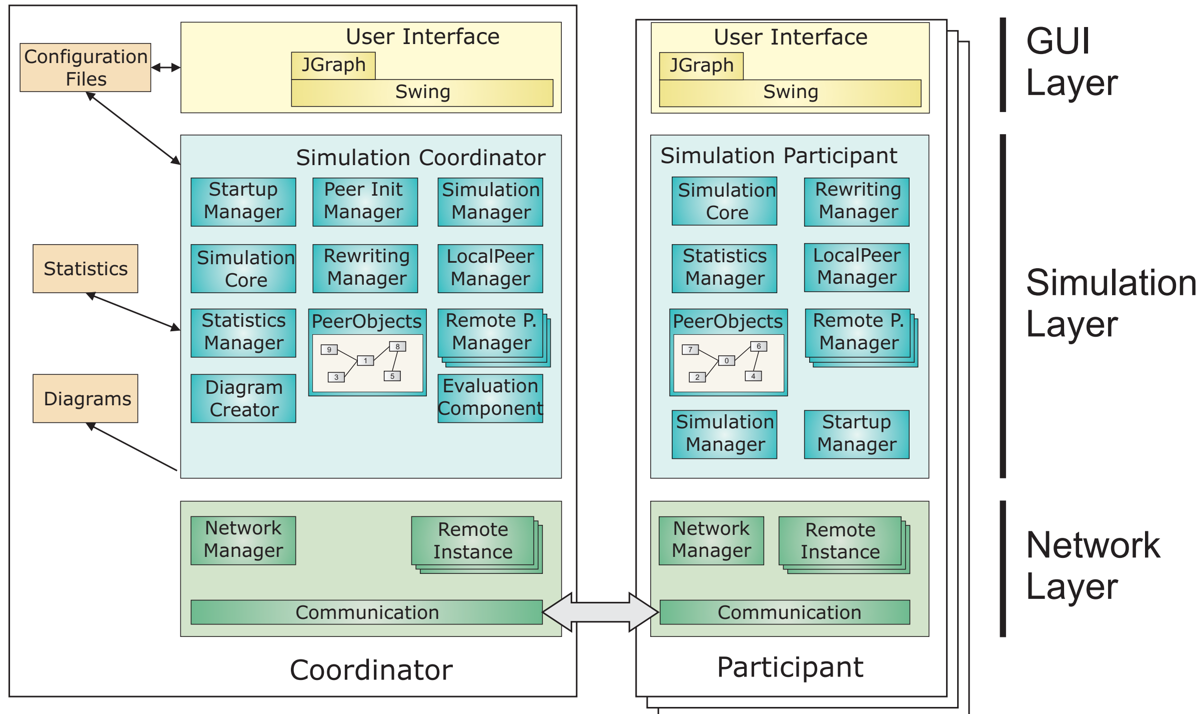


Figure 7.1: SmurfPDMS System Architecture

7.1.1 Network Layer

The network layer, which in our current implementation uses TCP/IP, is responsible for all communication between SmurfPDMS instances. Communication is necessary due to various reasons, e.g., finding further SmurfPDMS instances to participate in a simulation, communication between coordinator and participants to set up the initial configuration, synchronization, and collecting statistics at the end of a test run.

The network layer is independent from the simulation layer and all direct communication between SmurfPDMS instances is effectuated by the *Remote Instance* class that manages the connection to another SmurfPDMS instance. Thus, we can replace the communication protocol and still use the same interface without having to adapt the simulation layer. The only components in the simulation layer directly interacting with the communication layer, i.e., with Remote Instance, are the *Simulation Manager*, the *Remote Peer Manager*, and the *Remote Peer* – as illustrated in Figure 7.2 and discussed below.

7.1.2 Simulation Layer

The simulation layer incorporates all classes necessary to run a simulation. It contains the implementation of all techniques and strategies we have discussed in this dissertation (referred to as *Simulation Core*). Within the simulation layer we payed attention to allow extensibility so that a direct comparison between different approaches is possible. For instance, in this dissertation we have discussed and evaluated two types of distributed data summaries (QTree-based and histogram-based), but nevertheless SmurfPDMS can

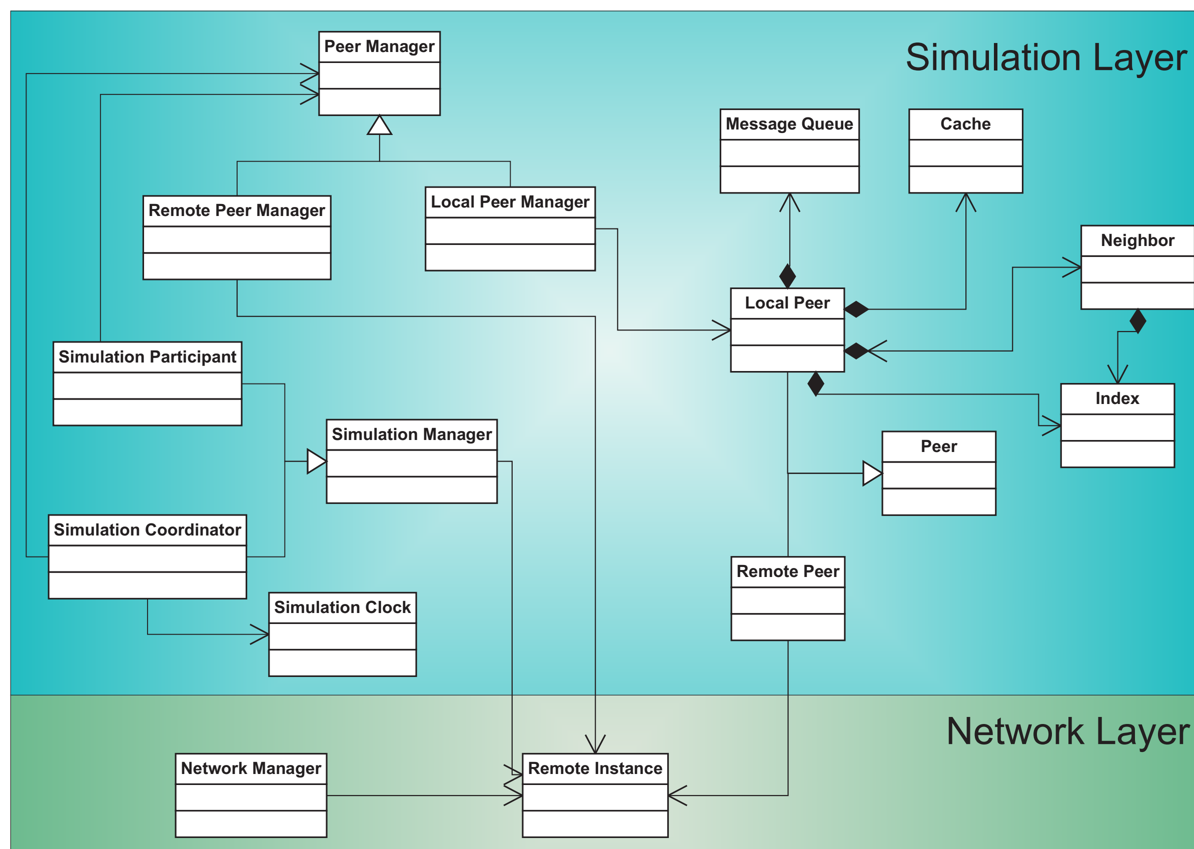


Figure 7.2: SmurfPDMS – Class Diagram: Associations between Simulation Layer and Network Layer

be extended by even more variants. This is also true for other techniques such as maintenance strategies, query processing strategies, and rewriting algorithms. In addition to extensibility we also set a high value on repeatability: that is why we enabled SmurfPDMS to retrieve aspects such as the simulated network and events from configuration files.

In addition to the Simulation Core, the simulation layer contains multiple managers (Figure 7.1) and components performing special tasks. The *Simulation Manager* implements the main routines of SmurfPDMS and is responsible for instantiating all other managers and running the simulation. It synchronizes SmurfPDMS instances and simulates query processing as well as dynamic behavior (Section 7.4.3). The *Startup Manager* and the *Peer Initialization Manager* implement all tasks necessary to start a simulation, e.g., determining the initial setup of the simulated PDMS, query load, partitioning the simulated network, and assigning the partitions to participating SmurfPDMS instances (Section 7.4.2).

The *Local Peer Manager* manages and simulates the behavior of the peers contained in the partition of the simulated network that has been assigned to a SmurfPDMS instance. As during a simulation it is necessary to exchange data between SmurfPDMS instances, e.g., queries might be sent from peers of the local partition to peers assigned to other partitions, the *Remote Peer Manager* transparently manages all communication to non-

locally simulated peers in conjunction with the network layer (Section 7.4.3).

While running a simulation the *Statistics Manager* collects statistics that, after the simulation has ended, can be used to create diagrams (using the *Diagram Generator*). In most cases the user may have to run multiple simulations to evaluate an approach properly. Thus, to enable the user to run such test series in an automated fashion, the *Evaluation Component* offers a variety of preconfigured test series, which can still be extended to support additional ones (Section 7.4.4).

7.1.3 GUI Layer

In order to make configuring and controlling a simulation more convenient, SmurfPDMS provides a graphical user interface [105], whose main window is shown in Figure 7.3. Using this GUI, the user can specify all configuration parameters necessary to run a simulation. To start a simulation the user selects some of the displayed SmurfPDMS instances and presses the start button (Section 7.4.1).

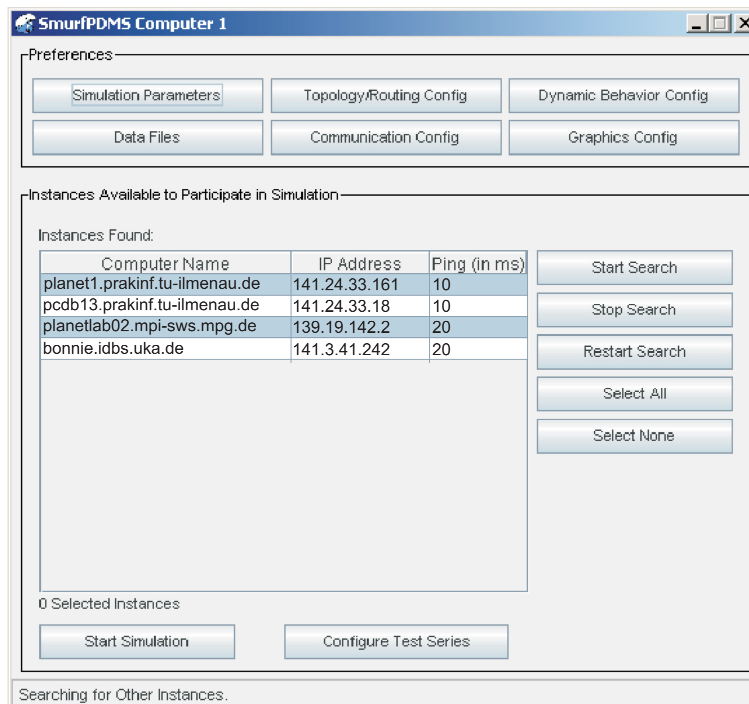


Figure 7.3: SmurfPDMS GUI – Main Window

Once the simulation is started, all GUI buttons are disabled for all participants. However, at the coordinator's site the *simulation window*, which enables the user to control the simulation, becomes available (Figure 7.4). It shows the simulated network of peers and the communication links between them. If a message is sent between any two peers, a message symbol is attached to the link. By double-clicking on a peer symbol, additional information about the peer such as the peer's local data, routing index, and mappings is displayed. Likewise, by double-clicking on a message symbol, additional information about the message is displayed, e.g., its type (answer, query, or update) and its contents (query POP tree, data records, ARLists, etc.).

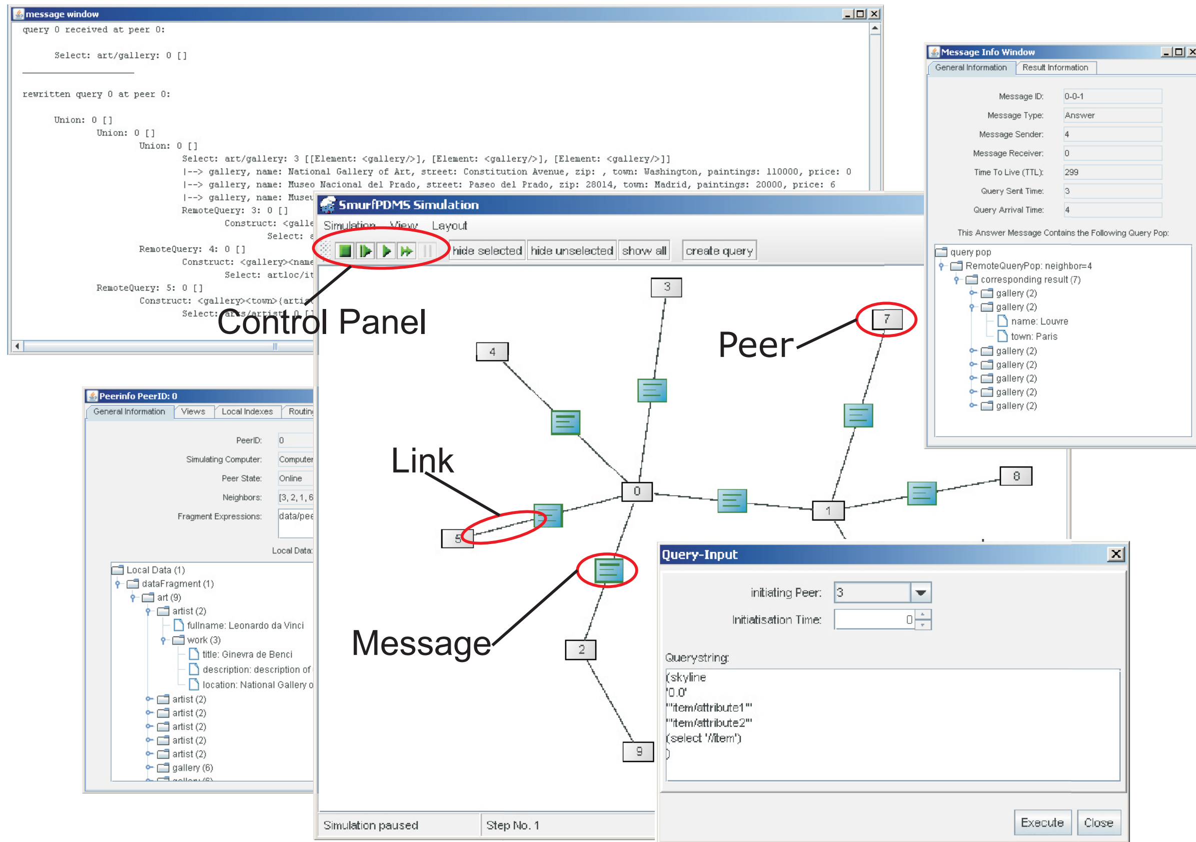


Figure 7.4: SmurfPDMS GUI – Simulation Windows: *Simulation Window (center), Rewriting Info Window (top left), Message Info Window (top right), Query Initiation Window (bottom right), and Peer Info Window (bottom left).*

In order to display the network topology we use JGraph [103], which is based on Java Swing. Although it provides a variety of features, its open source version provides only premature layout algorithms to display network graphs. Thus, we implemented two additional layout algorithms (Radial Tree [200, 207] and the Spring Model [67]), which we use in our current implementation to arrange the peers of the simulated network in the simulation window.

In addition to displaying the simulated network and the messages sent from one peer to another, the simulation window also provides a control panel. Its buttons allow the user to start, halt, or end the simulation without having to wait for any predefined stop criteria such as a maximum runtime for the simulation. Furthermore, the user is given the opportunity to decide if the simulation shall proceed automatically to the next simulation step once the previous is completed on all participants. An alternative is that the simulator processes only a user-defined number of steps and afterwards waits for the next user instruction. The user might also decide to issue a query. Figure 7.4 shows the corresponding window that enables the definition of a time step, a query, and a simulated peer to issue the query at.

If rewriting is activated, the user might want to access additional information about how a query is rewritten from one schema into another during runtime. Thus, Smurf-PDMS also provides a rewriting window (Figure 7.4) displaying the rewritings of pro-

cessed queries, which is updated whenever a peer rewrites a query. After the simulation has ended and the coordinator has collected all necessary statistics from the participants, some immediate statistics for the simulation are displayed to the user – Figure 7.5.

Figure 7.5: SmurfPDMS GUI – Statistics Window

7.2 Simulation Core

After having discussed the basic architecture of our system, let us now focus on how we integrated our techniques so that the simulator is still extensible. The main classes of the Simulation Core are sketched in Figure 7.6.

Routing Indexes and their Maintenance

Each simulated peer is represented by a *Local Peer* instance, which references one instance of *Neighbor* for each of the peer's neighbors in the simulated network. Each Neighbor instance unambiguously identifies a neighboring peer and has a reference to a routing index (instance of a class derived from *Compound Routing Index* – Figure 7.6). In Chapter 5 we have stated that in order to construct routing indexes each peer needs to compute a local summary of its local data (*Local Index*). Furthermore, this local summary plays an important role for index maintenance strategies and update merging. The need for both, a local index and a routing index, both using the same base structure, results in the class hierarchy depicted in Figure 7.6. Both classes Local Index and Compound Routing Index are derived from the same base class *Index*. Both are extended by QTree-based index

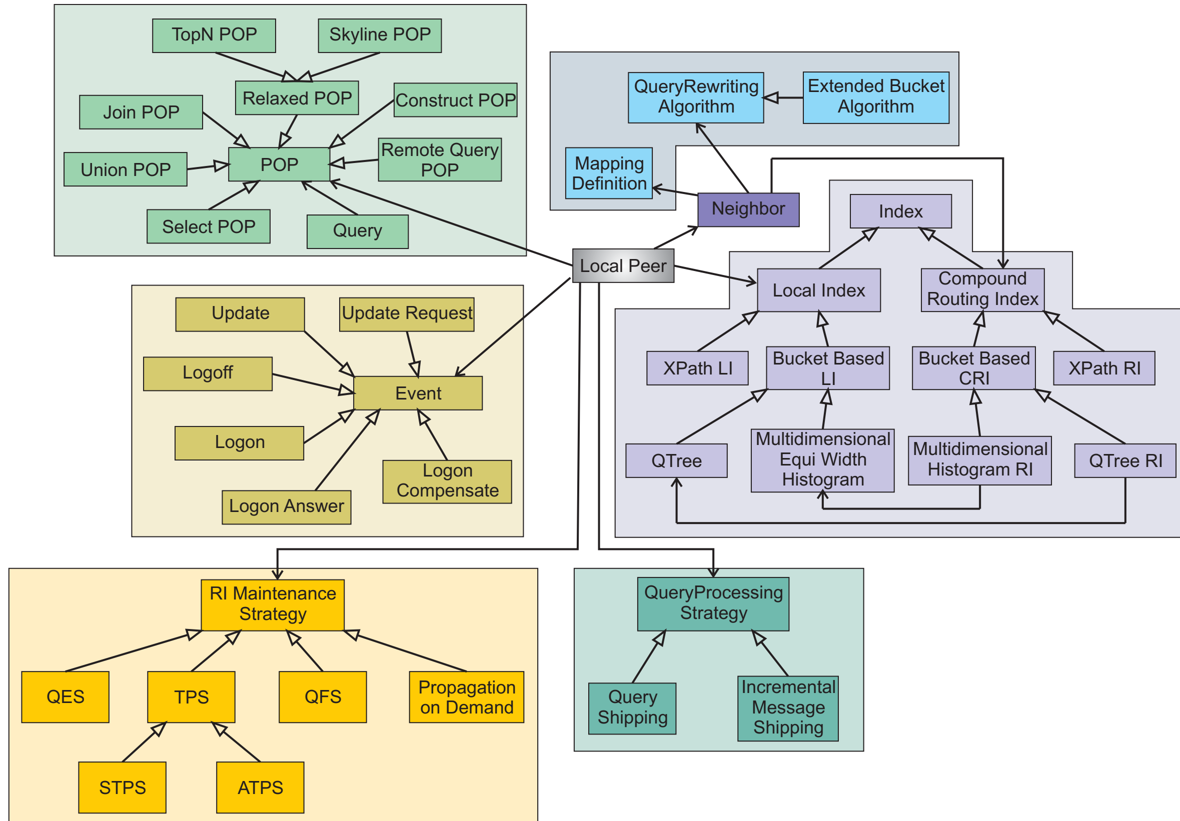


Figure 7.6: SmurfPDMS – Simulation Core Architecture. *For the sake of clarity, we display only the most important associations between the displayed classes.*

classes (*QTree*, *QTreeRI*), histogram-based index classes (*MultidimensionalEquiWidth-Histogram*, *MultidimensionalHistogramRI*), and XPath-based index classes (*XPathIndex*, *XPathRI*).

In the context of index maintenance, peers exchange different types of messages (Section 5.3) that are processed according to a maintenance strategy (*QES*, *STPS*, *ATPS*, *QFS*, *Propagation on Demand*). All these message types (*Update*, *UpdateRequest*, *Logoff*, *Logon*, *LogonAnswer*, and *LogonCompensate*) are derived from the same base class named *Event* and contain information the receiver peer of the message needs to process, e.g., a *Logon* message contains a local index summarizing the data of the peer that logs on.

Query Processing

As introduced in Section 3.3, queries in SmurfPDMS are represented as POP (plan operator) trees. These POP trees consist of algebraic plan operators encoding operations that are necessary to answer a query. Thus, our implementation contains one class for each supported plan operator (*SelectPOP*, *UnionPOP*, *JoinPOP*, *ConstructPOP*, *RemoteQueryPOP*, *TopNPOP*, and *SkylinePOP*, each derived from class *POP*). The algorithms that define how to process the input XML structures are part of these classes.

In Chapter 6 we have discussed how to exploit the result of a query's evaluation on a peer's local data in conjunction with routing indexes to identify relevant neighbors.

Furthermore, we have discussed how to use relaxation and constraints. We have implemented two strategies considering all these techniques: an incremental variant and a non-incremental variant (*Incremental Message Shipping* and *Query Shipping*).

Query Rewriting

In our current implementation there is only one strategy available to rewrite queries (*Extended Bucket Algorithm*). However, in order to rewrite queries, each Neighbor instance has not only a reference to a routing index but also a reference to a *Mapping Definition*, which encodes the mapping between the two peer schemas. According to the steps of query processing in SmurfPDMS (Section 1.2) a query is rewritten in the last step after having identified relevant neighbors. Thus, the Extended Bucket Algorithm is given the set of relevant neighbors (relevant on data-level) and the corresponding mapping definitions as input. The rewriting result is split up into remote queries, which are sent to the corresponding neighbors. As the answers to the queries are already rewritten into the schema of the rewriting peer, all further tasks are realized by the applied query processing strategy and not by the rewriting component.

7.3 Simulation Workflow

After having introduced the system architecture in the previous sections, this section gives on overview of the general steps a simulation adheres to (Figure 7.7). If multiple SmurfPDMS instances are participating in a simulation, the instance serving as coordinator fulfills several tasks that are crucial for running a simulation. The coordinator is determined implicitly at the beginning of a simulation – it is always that instance at which the user initiates the simulation.

Once the user has started the simulation, the coordinator invites the subset of detected instances, having been chosen by the user, to join the simulation. Afterwards, the coordinator determines the initial setup by reading a set of configuration files or receiving these parameters from the graphical user interface. This information might specify the complete configuration of the simulated PDMS or only a parametrical setup describing how SmurfPDMS shall create such a network. In any case, the configuration also comprises parameters for distributed data summaries, query processing strategies, routing index maintenance, etc. The coordinator also partitions the simulated network and assigns one partition to each participant and one to itself. When starting a simulation, all the parameters as well as the partitions are sent to the participants. While running the simulation, the coordinator, just as all other participants, simulates the behavior of its assigned partition and collects statistics. Furthermore, the coordinator synchronizes all participants and initiates events such as queries, peer crashes, and data updates.

A simulation automatically ends when either a maximum number of time steps (parameter of the coordinator) is reached or for a certain (configurable) number of time steps no peer has performed any actions and there is no message waiting in any message queue of any simulated peer. Alternatively, the user may end a simulation by clicking on the corresponding button in the graphical user interface. After the simulation ended, the coordinator collects the statistics from the participants and stores them along with its local statistics into a statistics file, which can be used to create diagrams later on.

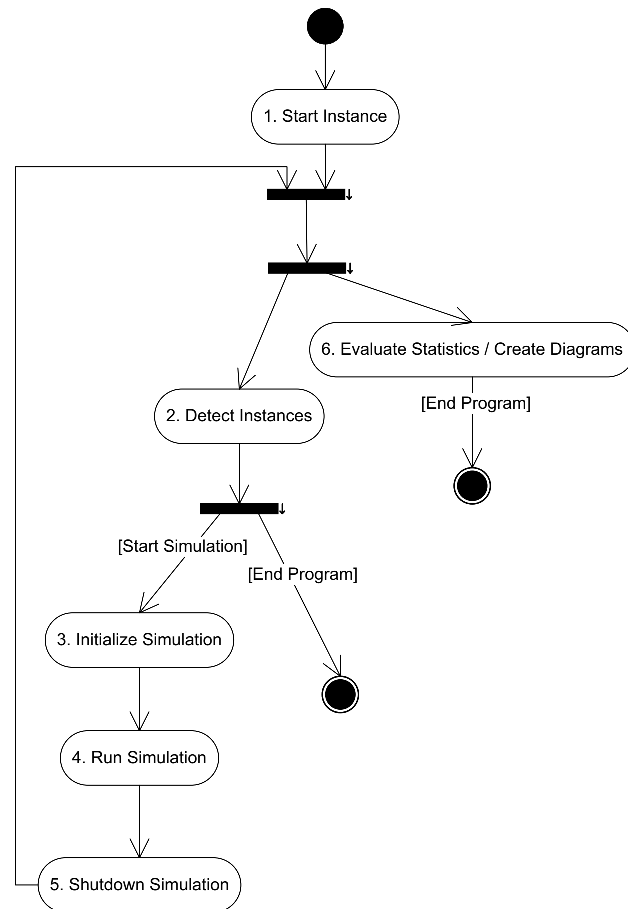


Figure 7.7: SmurfPDMS – General Steps of Running a Simulation

In summary, the most important tasks of the coordinator, distinguishing it from participants, are:

- invite participants to join the simulation,
- determine the initial setup and initialize participants,
- simulate a partition of the simulated network locally and collect statistics,
- synchronize all participants,
- choose queries and determine peers to initiate them,
- simulate dynamic behavior: local data updates, peer joins, and peer crashes,
- log events to ensure repeatability,
- end the simulation when a stop criterion is fulfilled,
- collect statistics from all participants,
- evaluate statistics and create diagrams, and
- shutdown the simulation by sending the participants a termination signal.

In contrast to the various tasks of the coordinator, participants only need to meet the following:

- answer an invitation to join a simulation,
- receive the configuration from the coordinator,
- wait for instructions from the coordinator and execute them (including synchronization), and
- simulate a partition of the simulated network locally and collect statistics.

7.4 Important Steps of Simulating a PDMS

So far we have only sketched the main steps a simulation adheres to. In this section we review some of these steps in more detail, these are: step 2 (detecting other SmurfPDMS instances), step 3 (initializing a simulation), step 4 (running a simulation), and step 6 (evaluating statistics and creating diagrams). Figure 7.8 illustrates the most important classes in the simulation layer that are instantiated at the coordinator and the participants to run a simulation.

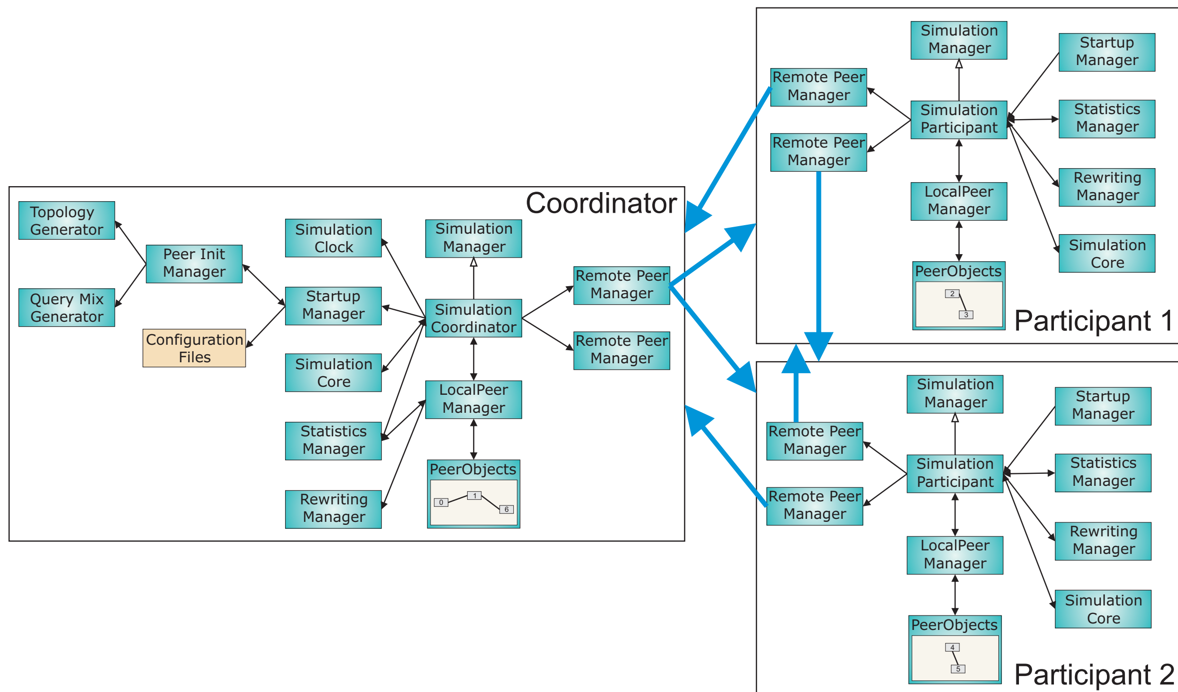


Figure 7.8: SmurfPDMS – Important Classes of the Simulation Layer Instantiated During Runtime. *Communication channels between SmurfPDMS instances are indicated by blue arrows.*

7.4.1 Detecting SmurfPDMS Instances

As mentioned above, SmurfPDMS uses several managers to group all actions that are necessary to perform a specific task. One of these managers is the *Network Manager* (Figures 7.1 and 7.2), which takes care of all tasks necessary to initialize communication between instances – step 2. Each SmurfPDMS instance is running only one Network Manager instance. Each Network Manager instance listens on a specific port for messages from other instances. It is instantiated with an IP address, a listening port, and an instance name.

Already on startup, a SmurfPDMS instance might know some other instances, either specified by the user or known from previous simulations. Thus, on startup the Network Manager tries to contact each of these instances. If an instance answers, both instances exchange their contact details (IP, name, and port) as well as details on all other already contacted instances. Furthermore, a new instance of Remote Instance (as already mentioned in Section 7.1.1) is instantiated, which will effect all direct communication with the newly-discovered SmurfPDMS instance.

All found SmurfPDMS instances are displayed in the graphical user interface (Section 7.1.3, Figure 7.3). The Network Manager periodically sends ping messages to all these instances and removes them from the list if they no longer answer. Based on the list of displayed instances, the user can finally select those instances that shall participate in the simulation. In addition to trying to contact instances directly, SmurfPDMS also supports multicast (given multicast address and multicast port), i.e., in regular time intervals the Network Manager sends a multicast message and adds all answering instances to the list that is displayed to the user and creates corresponding Remote Instance instances.

If, while running a simulation, the communication between coordinator and participant fails (after multiple attempts), the simulation coordinator aborts the simulation and sends abortion signals to all participants. As a consequence, the coordinator and all participants are reset so that a new simulation can be started.

7.4.2 Initializing a Simulation

Another important manager necessary to start a simulation is the Startup Manager. It takes care of starting the simulation – this includes starting all other managers. Although coordinator and participants both use the Startup Manager, it has to perform different tasks. In order to start and initialize a simulation (concerning step 3 of the simulation workflow introduced in Section 7.3), the Startup Manager at the coordinator has to:

- read all parameters from configuration files or receive them from the GUI,
- initiate collecting statistics (Statistics Manager),
- create the simulated network (Peer Initialization Manager),
- compute query load (Peer Initialization Manager),
- create Peer instances (Peer Initialization Manager),
- partition the simulated network and send the configuration to all participants,
- start all other managers necessary to run the simulation, and especially
- start the simulation by starting the coordinator's Simulation Manager instance: the *Simulation Coordinator*.

In contrast, the Startup Manager running at a participant only needs to:

- receive the configuration from the coordinator,
- initiate collecting statistics (Statistics Manager),
- start all other managers necessary to run the simulation, and especially
- start the local Simulation Manager instance: the *Simulation Participant*.

After having started the simulation, the Startup Manager at the coordinator's site takes care of determining the initial configuration. All necessary parameters can be read from configuration files or retrieved from the graphical user interface. Most of these parameters are one-valued, i.e., a string, a number, or a boolean value, indicating for example the threshold for an STPS propagation measure (Section 5.3.4). However, there are some aspects that, especially with respect to repeatability, cannot be configured that easily but have to be determined on startup: network topology and query load. Thus, in order to use the same simulated network or query load for multiple simulations, SmurfPDMS uses configuration files defining not only the topology of the simulated network but also aspects such as the data each peer provides and the set of queries issued during a simulation. Such configuration files can either be generated by external applications, manually by the user, or by SmurfPDMS itself. Therefore, SmurfPDMS knows several algorithms, we will sketch a few of them below. The produced configurations can be stored into configuration files and used for future simulations.

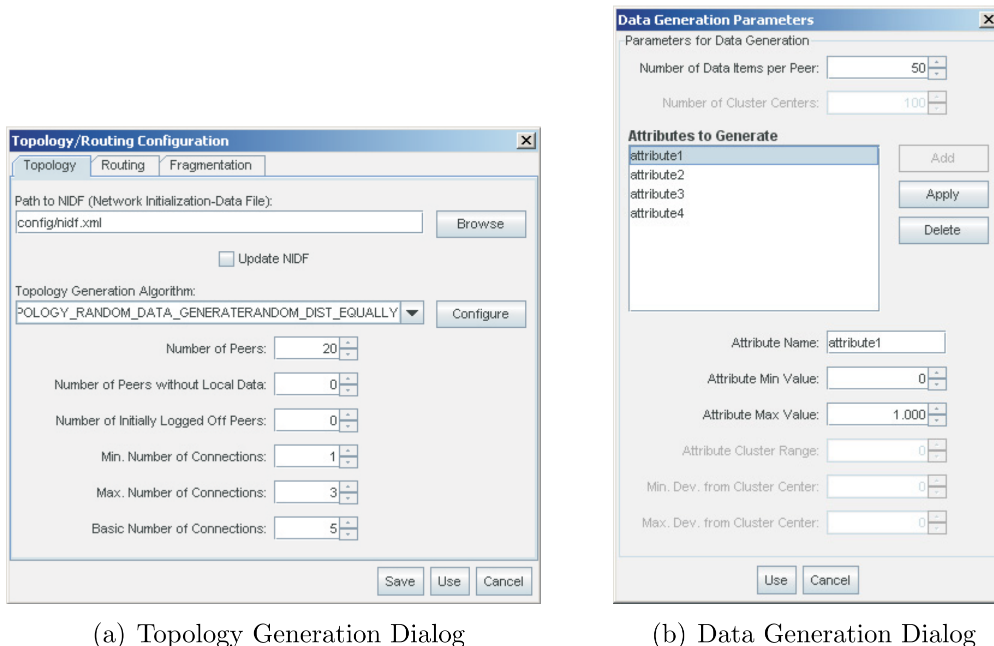
Topology/Data Generation

After having read the configuration files, the Peer Initialization Manager at the coordinator is started to create a network of peers. In order to create such a network, the Peer Initialization Manager first needs to determine the topology. If the network definition cannot be read from an existing configuration file, the Peer Initialization Manager needs to create a topology according to a set of input parameters – most of them are depicted in Figure 7.9. Some of these parameters are: the total number of peers, the minimum and maximum numbers of links per peer, and the number of peers that shall not provide any local data. Furthermore, the user can specify that a certain number of peers shall not be connected to other peers on startup so that they might join the network later on when the simulation is running. In addition to these parameters, the user can choose between several algorithms to create topologies with different characteristics.

The first distinction with respect to these algorithms is the choice whether the network should be cycle-free or not. Another distinction concerns the data assigned to the peers: the data can be generated or an existing data file can be read so that its content can be distributed among all simulated peers. If the data is to be created by the simulator, the user needs to decide what kind of data shall be represented by each peer: in addition to the number of data records per peer, also the number of attributes per record and their value ranges are defined by parameters. In the simplest case, the data records are generated randomly in the defined data space and assigned to the peers. However, the Peer Initialization Manager, given some additional parameters, can also generate all the other distributions we have used to evaluate our techniques in Section 6.5: clustered data in a clustered distribution, clustered data in a random distribution, and anticorrelated

data. If the data is not to be generated but read from an input file (XML format, processed using Saxon[2]), the input data is split up into fragments by applying alternately vertical and horizontal fragmentation so that the obtained fragments can be assigned as local data to the peers. The data of each peer is defined by XPath expressions that unambiguously designate a portion of the data file exclusively represented by the peer.

However, although this works fine for creating setups resembling unstructured P2P networks, these algorithms do not yet consider heterogeneity. Thus, neither peers with different local schemas nor corresponding mappings are created automatically. Nevertheless, such networks can still be used in conjunction with SmurfPDMS if their specifics are defined in configuration files created by an external component or manually by the user.



(a) Topology Generation Dialog (b) Data Generation Dialog

Figure 7.9: SmurfPDMS – Topology and Data Generation

Query Load

In order to simulate query processing, the coordinator needs to determine a set of queries to be issued during runtime. This set of queries can either be read from an existing configuration file or determined automatically on startup by the Peer Initialization Manager at the coordinator. Only queries contained in this initial query load can be issued automatically during runtime. Configuration parameters (some of them are illustrated in Figure 7.10) determine when which query is issued at which peer.

In order to generate the query load on startup, the Peer Initialization Manager considers another set of parameters specifying the composition of the query load, e.g., these parameters could specify that the query load should contain 3 level-1-queries (query POP trees with only one level, i.e., queries consisting of only one select POP) and 3 level-2-queries (for instance, a query consisting of a select POP and a construct POP). If it is

[2]http://saxon.sourceforge.net/

possible to load all necessary queries from the configuration files, these queries are used as query load for the simulation without having to generate additional ones. If there are not enough queries in the configuration files, the Peer Initialization Manager needs to create some additional queries to meet the specifications. Additional level-1-queries are created by taking the schema of the peers' data into account and defining select POPs with appropriate XPath expressions. Creating queries of higher levels is more difficult. As defining sensible joins, top-N, and skyline POPs automatically is complicated, we apply a straightforward heuristic that allows for creating queries of arbitrary levels: we simply combine two level-1-queries by means of a union POP to obtain a level-2-query, we combine two level-2-queries to obtain a level-3-query, and so on.

However, the query load generator is used only rarely because in most cases the user wants to evaluate specific queries that he/she specifies in advance. By specifying both queries and initiating peers in advance in the configuration files, it is possible to selectively issue queries at specific peers.

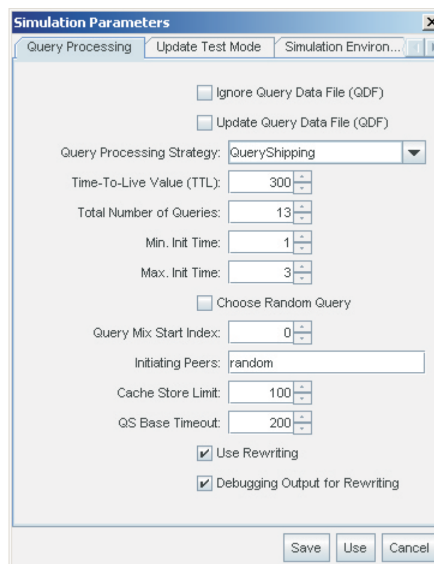


Figure 7.10: SmurfPDMS – Query Processing Parameters

Determining Partitions

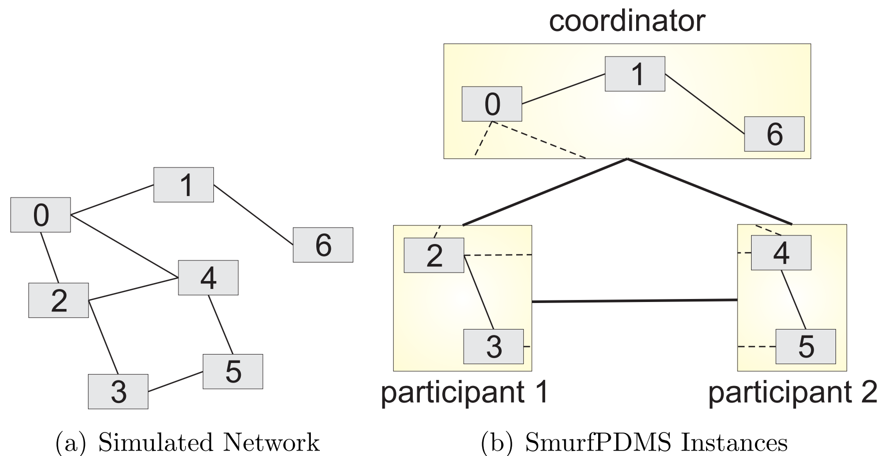
As we are working with XML data and a Java implementation, handling such data requires much main memory space. In order to share resources, especially main memory, we decided to have multiple instances of SmurfPDMS, running on different machines, collaborate and share the load. For this purpose, the coordinator's Startup Manager has to partition the PDMS we want to simulate into several partitions and assign each partition to an instance.

In principle, we could randomly assign peers to instances. However, this solution would likely result in a high communication overhead between instances. As peers only send messages to neighboring peers, the partitioning algorithm should consider these relationships. Thus, the algorithm used by the Startup Manager tries to group neighboring

peers within the same partition. Each created partition has approximately the same size: $\frac{\#peers}{\#participants+1}$

Figure 7.11 shows an example of partitioning peers. The peers of the simulated network (Figure 7.11(a)) are to be assigned to three SmurfPDMS instances (coordinator and two participants). All participants have direct communication links to the coordinator (Figure 7.11(b)). Furthermore, in dependence on the connections between simulated peers, instances might also communicate directly with each other to exchange messages, e.g., query messages from peer 3 to peer 5 are sent from participant 1 directly to participant 2 without taking the detour to the coordinator.

In our current implementation, each instance needs to receive the initial setup (including the partition and the parameters) from the initiator. In future work we plan to extend our system with a decentralized startup procedure so that the coordinator's load is reduced.



(a) Simulated Network (b) SmurfPDMS Instances

Figure 7.11: SmurfPDMS – Communication Connections between SmurfPDMS Instances

7.4.3 Running a Simulation

After having completed the startup process, the *Simulation Coordinator* begins the actual simulation (step 4), which simulates the behavior of the simulated PDMS in the presence of queries and dynamic behavior (local data updates, peer joins, and crashes) – requiring communication between coordinator and participants.

Issuing Queries

We have already mentioned above that the query load, i.e., the set of queries issued *automatically* by the system, is determined already on startup. The coordinator chooses a peer, a query, and a time step and "orders the peer" to issue a query. Queries, peers, and time steps can either be specified in configuration files to achieve repeatability of simulations or the coordinator can be configured to randomly choose a query and a peer and have it issue the query in an arbitrary time step. Alternatively, the user might use the graphical component to issue a query, which does not have to be predefined in a configuration file, during runtime.

Dynamic Behavior

The simulation of dynamic behavior is controlled by a set of parameters – the subset concerning data updates is shown in Figure 7.12. Alternatively, these events can also be read as input from a configuration file created by a previous simulation.

In order to have SmurfPDMS induce dynamic behavior, the user can for example specify a probability that a peer updates its local data in a time step. Moreover, he/she can limit the number of peers allowed to update their local data per time step and in total. Further parameters controlling dynamic behavior are the number of local data records that are to be updated and the definition to what extent attribute values shall be changed (assuming all peers adhere to the same schema). As an alternative to having the coordinator randomly decide to what extent a data record shall be changed, SmurfPDMS provides an alternative: the user can specify a second preconfigured network setup that uses the same topology but assigns different data to the peers. While running the simulation, the data distribution is slowly converted from the first into the second distribution according to the user-defined parameters.

Figure 7.12: SmurfPDMS – Data Updates Parameters

Synchronization

While running a simulation, communication between simulation coordinator and participants is necessary, for instance, for synchronization. Figure 7.13 illustrates which classes

in the simulation layer and in the network layer realize communication. The coordinator needs to signal all participants to proceed to the next time step, i.e., to compute one of the received messages for each simulated peer. To make this as transparent as possible, the coordinator holds a *Peer Manager* instance for each participating SmurfPDMS instance (including itself) and invokes the same method at each Peer Manager. In this context, we distinguish between Local Peer Manager and Remote Peer Manager – both derived from Peer Manager. Each SmurfPDMS instance holds exactly one Local Peer Manager instance with references to all Peer instances (Local Peer) – representing peers that are part of the partition of the simulated network simulated locally – and multiple Remote Peer Manager instances for peers simulated by other SmurfPDMS instances.

Because of the inheritance, the coordinator only needs to call the same method at each Peer Manager in order to have each SmurfPDMS instance proceed to the next time step. Then, the Peer Managers take care of all further actions that need to be performed in order to proceed to the next time step. A Local Peer Manager instance can easily realize message processing for each of its Local Peer instances as it holds direct references to them. In contrast, a Remote Peer Manager needs to communicate with another SmurfPDMS instance, or with the Local Peer Manager of another SmurfPDMS instance respectively. In order to realize this kind of communication, each Remote Peer Manager holds a reference to the Remote Instance in the network layer that enables communication to the SmurfPDMS instance whose peers it represents. Thus, when the coordinator signals to proceed to the next time step, a Remote Peer Manager at the coordinator contacts the corresponding SmurfPDMS instance via the Remote Instance, which in turn takes care of transmitting the signal to the SmurfPDMS instance (via its Remote Instance, its Simulation Participant, and its Local Peer Manager). Figure 7.13 illustrates this process and the classes involved.

A simulation step is completed when all participating SmurfPDMS instances have completed the simulation step. The coordinator starts the next simulation step once all participants have signaled the completion of the previous step.

Exchanging Messages between Peers

As discussed above, when the coordinator asks a participating instance to proceed to the next time step, the participant's Local Peer Manager instance computes one message for each peer simulated locally. To process these message, we use a thread-based solution so that the overall processing time at the corresponding computer may be reduced. However, processing messages for simulated peers often entails sending messages from one simulated peer to another. "Sending" such messages is managed by the Simulation Manager by identifying the Peer Manager that is "responsible" for the receiver peer. If that Peer Manager is a Local Peer Manager, no communication between SmurfPDMS instances is necessary. If the responsible Peer Manager is a Remote Peer Manager, the message is transferred via the Remote Instance of the communication layer to the corresponding SmurfPDMS instance. Upon reception and deserialization by the Simulation Manager at the receiving SmurfPDMS instance, its Local Peer Manager is instructed to "deliver" the message to the receiver peer. Hence, the procedure of sending a message from one simulated peer to another follows the pattern illustrated in Figure 7.13.

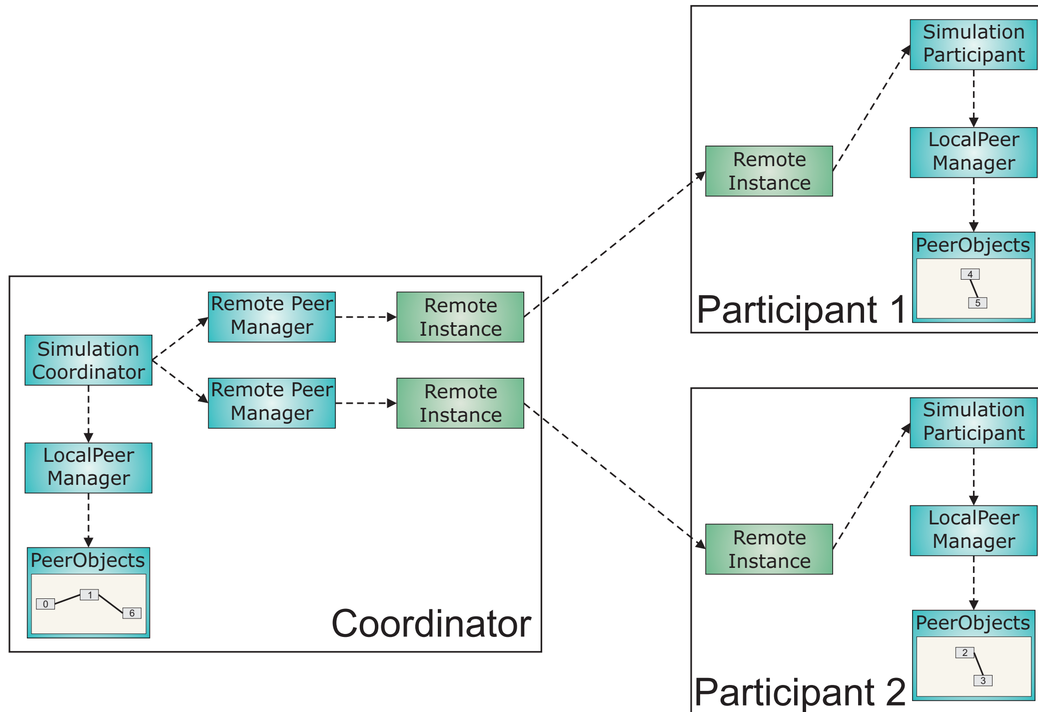


Figure 7.13: SmurfPDMS – Interactions between Classes for Synchronization

Retrieving Up-to-Date Information for the GUI

Still, there is another situation that requires communication between SmurfPDMS instances. The graphical user interface at the coordinator provides the user with a simulation window (Section 7.1.3), which displays detailed information about the current state of the simulated network. In order to provide the user with this kind of information, it is necessary to retrieve those details during runtime from the SmurfPDMS instances that simulate the peers. To make this as transparent as possible to the GUI layer, the coordinator holds *Peer* instances (Figure 7.2) for each peer being part of the simulated network (including both *Local Peer* and *Remote Peer* instances, the latter representing peers simulated by participants). The simulation window implementation does not need to distinguish between Local Peer instances and Remote Peer instances as both provide the same interface for retrieving information about the simulated peer. If detailed information, for instance about a peer's current entries in its message queue, is requested, the coordinator can easily obtain that information from the peers simulated locally by directly accessing the Local Peer instance. If detailed information is requested for a peer simulated by another SmurfPDMS instance, i.e., if the same method is called at a Remote Peer instance, the request is forwarded to the SmurfPDMS instance simulating the peer. In doing so, the request is forwarded to the correct SmurfPDMS instance (via its Remote Instance, Simulation Participant, and Local Peer Manager) so that the requested data can be retrieved from participants and displayed to the user.

7.4.4 Evaluating Statistics and Creating Diagrams

We created SmurfPDMS to help us evaluate our approaches. Collecting statistics, creating diagrams, and running test series are important and useful features whose details are discussed in the following.

Collecting Statistics

In order to use statistics after having run a simulation or a test series to create diagrams, they have to be collected during runtime by the Statistics Manager. SmurfPDMS distinguishes between a variety of different statistics that are collected in dependence on the configuration. For tests of query processing strategies it is, for instance, adequate to remember the number and the volume of messages sent between peers in order to process a query. For other tests, e.g., for tests with respect to index maintenance strategies, we need to measure correctness and the network traffic caused by the updates instead of query load. Thus, in dependence on the aspect we want to examine, we need to collect different kinds of statistics and configure SmurfPDMS accordingly.

On startup, each participant is informed by the coordinator what statistics to collect locally. After the simulation, the coordinator collects all statistics from all participants, aggregates them if necessary and stores them into a statistics file on disk. As we discuss below, these statistics can be used to create diagrams later on.

Running and Configuring Test Series

SmurfPDMS can be configured to run multiple tests in a row without requiring any user interaction. With respect to the activity diagram of Figure 7.7 this means that, once a simulation ends, all participating SmurfPDMS instances go back to the initial state and the coordinator automatically proceeds to the next simulation. In oder to accelerate the simulations, the *simulation window* of the graphical user interface (Section 7.1.3) is switched off as it would only unnecessarily slow down the simulation progress.

Running a test series means that the majority of parameters are the same and only a few are adapted to examine their influence. Figure 7.14 shows an example dialog that configures a test series to evaluate the influence of the relaxation parameter ε on processing a specific top-N query. In this dialog the user specifies ε for the first test run and the last test run as well as by what value to successively increase ε for all other test runs in between. This dialog further provides the option to vary between routing index implementations (QTree and multidimensional equi-width histogram) and different numbers of buckets. This means for each specified number of buckets for the QTree and for each specified number of buckets for the histogram one test series is run varying ε as described above.

Generating Diagrams for Test Series

After having completed all test runs, SmurfPDMS also provides several algorithms that use the collected statistics of all test runs as input to create diagrams illustrating the influence of the varied parameters. In dependence on the test series, the diagram generator

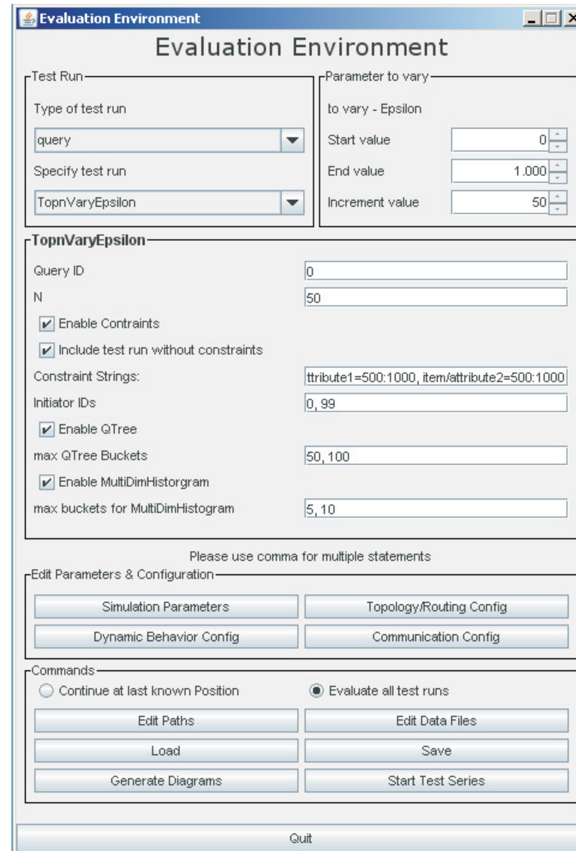


Figure 7.14: SmurfPDMS – Evaluation Environment

component (Figure 7.15(a)) offers applicable diagram types and initiates the computation of those that are chosen by the user. The standard output format is either Gnuplot[3] or Asymptote[4]. Both output formats are by default automatically converted to PS or PDF.

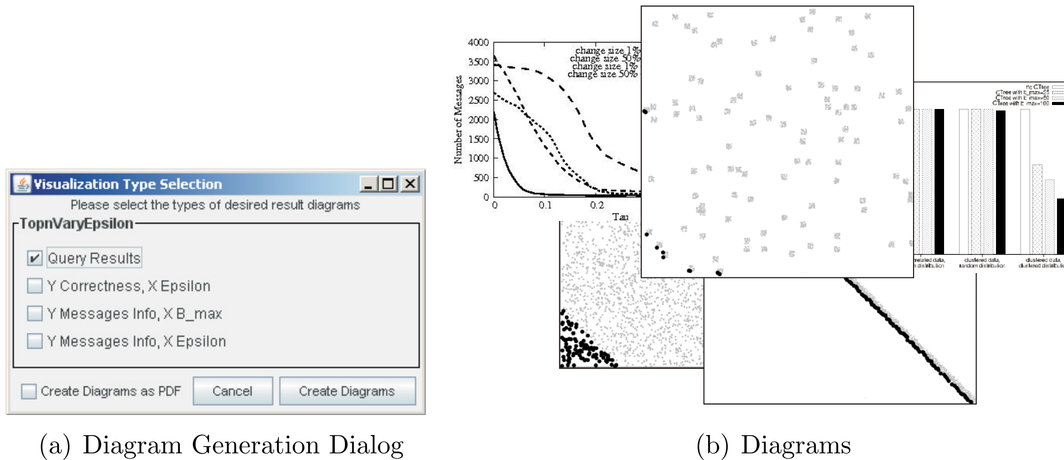
Figure 7.15(b) illustrates some example diagrams created using the diagram generator. Besides, most diagrams in this dissertation, which we used to illustrate our evaluation results, were generated by this component.

7.5 Evaluation

In order to illustrate the benefits of sharing the simulation load among multiple computers, we illustrate the results of some tests with up to 5 computers – all with the same hardware setup (1 GHz Pentium IV processors, 1 GB main memory) running SuSE Linux 10.0 (kernel v. 2.6.13-15.16) and Java v. 1.6.0_07. We simulated the behavior of a network of 1000 peers with cycles – each peer having 5 to 10 neighbors. We chose a simple setup for our evaluation, i.e., all peers in the network had the same schema and we did not simulate dynamic behavior. The query load consisted of 6 queries (simple selection queries) and was issued at the same time steps at the same peers in each simulation.

[3] http://www.gnuplot.info

[4] http://asymptote.sourceforge.net



(a) Diagram Generation Dialog (b) Diagrams

Figure 7.15: SmurfPDMS – Diagram Generation

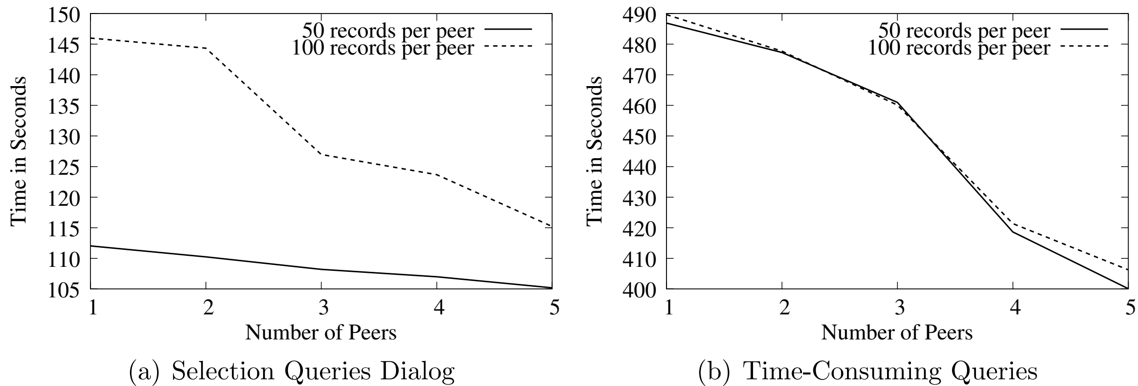
In addition to the number of computers, we varied the number of records provided by each peer (50 and 100, each record consisting of 4 random value numerical attributes restricted to the same ranges).

For each simulation we measured the absolute time the simulation was running (corresponding to 100 simulated time steps). The results we obtained are illustrated in Figure 7.16(a). For both configurations (50 and 100 records per peer) we see that by sharing load among multiple computers, the total simulation time can be reduced. Obviously, one factor that influences the computational load caused by running a simulation is the amount of data a query has to be evaluated on. If multiple peers share this load, the absolute time it takes to run a simulation is reduced.

Another factor influencing query load is query complexity. Figure 7.16(b) shows the results we obtained for simulating the same networks while issuing more complicated queries (again six). These queries make use of all operators supported by our system, i.e., select-project-join queries, relaxed skyline and top-N queries, and also queries involving union and construct operators. Our results show that evaluating such queries on the data increases the absolute simulation time by a factor of almost four. In comparison to Figure 7.16(a) the difference between the two setups (50 and 100 records provided by each peer) is smaller. Nevertheless, the basic tendency that sharing load among multiple computers reduces absolute simulation time remains. Of course, if there are too many computers participating in a simulation, the communication overhead between them counteracts the benefits so that the simulation time cannot be reduced by adding further computers.

7.6 Conclusion

In the beginning of this chapter, we have formulated several requirements that a simulating environment should fulfill. Due to its architecture and the use of inheritance and appropriate base classes, SmurfPDMS fulfills them all and can easily be extended by future strategies with respect to query rewriting, query processing, approximation, POPs, routing indexes, and index maintenance. SmurfPDMS supports XML as native



(a) Selection Queries Dialog (b) Time-Consuming Queries

Figure 7.16: SmurfPDMS – Performance of Distributed Simulations

data format and is implemented in Java so that it is platform independent.

Each simulation is repeatable since the initial setup as well as dynamic behavior and issuing queries can be configured with reusable configuration files. This also enables the user to examine the influence of individual parameters. Furthermore, SmurfPDMS provides a set of tools helping the user in creating configurations. These configurations support arbitrary network topologies that can be created by SmurfPDMS, by an external component, or manually by the user. As simulating the behavior of large networks might exceed the capacities of a single computer (or increase the time it takes to run a simulation), we decided to allow multiple computers share their resources – leading to the distinction between coordinator and participants. Let us again emphasize that the coordinator does not correspond to a mediator in a data integration system but only synchronizes participating SmurfPDMS instances and ensures repeatability of a simulation.

The ability to run whole test series without requiring any user interaction is another feature helping the user in conducting experiments. Collecting statistics during runtime and providing routines to create diagrams automatically is another feature that distinguishes SmurfPDMS from other systems. We used these features to conduct most of the experiments that evaluated the techniques proposed in this dissertation. Thus, we have already implemented a variety of test series and diagram generators, which can easily be extended in the future. The 3-layered architecture ensures independence from network communication and the graphical user interface. The latter, however, provides a handy tool to monitor and control a simulation in a very comfortable manner. Finally, let us mention that we can completely switch off the rewriting component so that SmurfPDMS can also be used to simulate the behavior of unstructured P2P networks in which all peers use the same schema.

Making SmurfPDMS a Real Application

So far we have only considered how to run a simulation. As this is exactly what we needed to evaluate the techniques discussed in this dissertation, there actually was no need to implement a real application. However, in order to turn SmurfPDMS into a real application, we need to dispose of the distinction between coordinator and participant. Furthermore, so far each SmurfPDMS instance represents multiple peers that participate

in the simulated network. In a real application, each instance has to represent only one peer and the network topology has to be determined implicitly by the connections between participating computers.

Especially the tasks that distinguish a coordinator from a participant are no longer required, e.g., synchronization and discrete time steps are no longer necessary. Furthermore, aspects such as initial configuration, network topologies, dynamic behavior, and issuing queries do no longer have to be propagated and controlled by the coordinator. Instead, each participant has to be able to decide on its own when it issues queries or leaves the network. Thus, the startup procedure has to be realized in a decentralized fashion so that the provided data is not determined by a coordinator but by the computers providing it.

Hence, in order to turn SmurfPDMS into a real application, we need to cut down on the functionality that SmurfPDMS provides for simulations, but the basic implementation, especially of the Simulation Core, could still be used. However, although SmurfPDMS started as a simulation environment and as it incorporates several specifics and techniques that a real application would also have to support, we refer to the resulting PDMS variant as SmurfPDMS – referring not only to the simulation environment but also to the whole concept of a PDMS that incorporates the techniques presented in this dissertation.

Chapter 8

Conclusion and Future Work

As we have seen in this dissertation, the PDMS approach is a promising technique to share data amongst autonomous peers. The underlying network model can be considered as an unstructured P2P system, which means there are no requirements on the topology and peers are free to choose any subset of participating peers as neighbors in the PDMS overlay network. Our aim was to extend PDMS functionality to support rank-aware query operators and to enable efficient query processing of queries involving such operators based on XML data and LAV-style mappings.

However, the autonomy, especially schema heterogeneity, of peers in a PDMS entails several complications with respect to query processing. One of them is the need to rewrite queries so that in order to answer a particular query formulated in a peer's schema, data provided by other peers in the system can be considered. This entails two immediate problems: how to describe correlations between the data provided by peers using different schemas and how to rewrite the queries efficiently. The answers to these problems strongly depend on the data model and the query language. The solutions we have proposed work on XML data and consider queries on an algebraic level. Consequently, each query language that allows for formulating the supported operations can be used. Using LAV-style mappings to encode correlations between schema elements, we extended the Bucket Algorithm to support queries containing rank-aware operators and XML data.

As we assume PDMSs to be based on unstructured P2P networks of autonomous peers, there is no central component we could use for query optimization. This means that global reasoning in the context of query rewriting is impossible. In contrast, each peer has to optimize a query with only the information being available locally. Although we can add additional information to query messages (e.g., a query ID), which might help identify cycles, it is possible that a peer is queried multiple times concerning the same original query. As a consequence, the result possibly contains duplicates and query execution costs might be higher than actually necessary.

In large networks with high numbers of peers providing data, queries may have a large number of result records. The user, however, is in most cases only interested in a small subset of them. A good option to make the result set more precise is to rank the result tuples according to a user-defined ranking function and display only the top N result records. Sometimes the user is not able to define just one ranking function but wishes the result to be ranked by multiple criteria. Thus, given multiple ranking functions skyline queries are an option that should be available to the user. Both kinds

of queries offer a great potential for optimization because if only a small portion of the complete result set is actually relevant to the user, there is no reason to retrieve all the other irrelevant records.

In order to exploit this observation for distributed query processing, a peer can add additional information to the query about highly ranked records that are already known to the peer. The receiver peer of such a query can use the additional information to send only those records with an answer message that are ranked better. Although this already reduces network traffic, another optimization goal must be to reduce the number of peers involved in answering a query. Thus, a peer needs to identify relevant and irrelevant neighbors.

Our solution considers two techniques a peer can use to identify relevant neighbors. First, a query is only forwarded to a particular neighbor peer if its schema has correspondences to the schema elements referred to by the query (schema-level). Second, a query is only forwarded to a neighbor if it provides data that is relevant to the query with respect to the attribute values (data-level). Whereas the first technique is a direct consequence of applying a query rewriting algorithm, the second technique requires further knowledge about the data stored at a neighbor that exceeds schema correspondences but describes the attribute values of records accessible via the neighbor.

Routing indexes, or distributed data summaries respectively, are appropriate structures that provide such information. In the context of this dissertation, we have identified histograms (QTrees) as a good base structure for routing indexes as they fulfill several beneficial requirements. Especially the QTree summarizes records with similar attribute values with a low approximation error. Furthermore, because of their multidimensionality, distributed data summaries capture attribute correlations so that the number of false positive routing decisions can be minimized. Further important fulfilled requirements are the support of efficient lookups and the fact that these structures are not only beneficial for rank-aware queries but also useful for other query types such as range and selection queries.

Routing indexes are well-known in P2P systems. Their application in the context of PDMSs is a contribution of this dissertation. We have proposed efficient strategies exploiting routing indexes to identify neighboring peers that are relevant to a particular query. In this context, we focused on rank-aware query operators and extended our considerations to support approximation. The approximate query processing techniques we propose reduce query execution costs even more by representing data provided by a neighbor with data records a peer has already processed. The degree of applied approximation is defined and limited by the user so that the obtained results meet the user's needs.

Another consequence, resulting from peer autonomy, is that peers are free to update their local data or leave/join the network at will. Consequently, routing indexes might become out-of-date. Since efficient techniques for query processing rely on the information provided by the routing indexes, we have to apply appropriate maintenance strategies to keep them up-to-date. We have discussed several approaches which might be used to achieve this goal. The bottom line is that we have to find an acceptable tradeoff between network load and freshness. In favor of freshness we in general prefer update propagation strategies that actively propagate updates to their neighbors according to some user-defined parameters with the aim of minimizing the total amount of network

load caused by the propagation.

In summary, in this dissertation we have extended PDMS functionality to support rank-aware queries. In particular, to the best of our knowledge we are the first to having discussed how to efficiently process rank-aware queries in PDMSs using routing indexes and relaxation. Furthermore, we have proposed a novel variant of routing indexes as well as several maintenance strategies to keep them up-to-date. To the best of our knowledge, we are the first to consider the maintenance problem on a level that goes beyond reconstruction. Moreover, we are not aware of any related work that applies routing indexes and approximation in PDMSs and unstructured P2P networks. Finally, we have also proposed an extension of the Bucket Algorithm considering rank-aware operators and referring to XML data.

Future Work

Regarding future work we identify three main sectors:

1. optimization and incorporation of state-of-the-art techniques
2. loosely-coupled data integration
3. uncertain data

We have already mentioned several extensions and optimizations of the techniques presented in this dissertation, e.g., supporting non-rectangular bounding boxes for the QTree, indexing string and numerical attributes, integration of GLAV-style mappings, etc. Furthermore, aspects that are out of the focus of this dissertation could be considered by future work, e.g., integrating security policies to establish mutual trust or improvements of cycle detection in conjunction with query rewriting. Especially the integration of state-of-the-art techniques for distributed query processing exceeding optimizations for rank-aware query operators is an interesting aspect. Although our strategies so far already support join queries, they do not yet provide any specialized optimizations that the literature proposes.

One of the problems we have to deal with in our current implementation is that pruning a peer on data-level based on routing indexes is very difficult in the presence of a join because estimating the join cardinality is not straightforward. A join is always formulated on two base structures of a peer's local schema. As the data is not stored in a joined format, the join attributes are not indexed by the same summarizing structure. Thus, the routing index can only provide information about each subgoal (base structure, e.g., libraries and books) in isolated summaries. Because of the separation it is difficult to estimate the result cardinality as there is no information about correlation of the join attributes. Thus, in our current implementation we only prune peers on data-level in conjunction with join queries if a peer does not provide any data with respect to the join attributes.

This can be optimized in future work by considering the data provided by each peer with respect to each subgoal in detail. Assume we have two peers, each peer provides data with respect to only one subgoal that the join is defined on. Because of the routing indexes we know approximations of the attribute value distributions of the attributes the join is defined on. Based on this knowledge we can estimate the number of join result records – we can even determine if it is possible that the join result yields any

result records at all. We can exploit this knowledge to prune additional peers from consideration or in an advanced "best-effort" strategy we could even use the estimated result size and the expected execution costs to decide whether it is worthwhile to compute the join between the data of a particular pair of peers or not.

Additionally, we could consider strategies for join processing used in distributed database systems in general [150]. For example, a join between data stored at different peers does not necessarily (although it is our current implementation) have to be computed at the peer that rewrites the query but it can also be computed at one of the two neighbors whose data is involved in a rewriting containing the join or even at another peer in the system. The optimization algorithm could, for instance, decide to transfer the smaller data set to the peer holding the bigger data set so that only one instead of both data sets has to be transferred to the rewriting peer. Further optimization is possible by considering options such as ship-as-whole or fetch-as-needed.

To transfer the local data of one peer to another in the presence of schema heterogeneity, we can exploit the fact that rewritings contain query snippets. The construct POP of a query snippet describes how to transform the data of a neighbor into the schema of the rewriting peer. If both neighbors transform their data into the schema of the rewriting peer, one of these peers can transfer its transformed data to the other peer that can now compute the join as both data sets now adhere to the same schema. In addition to the computation of the join operator, further operators of the rewritten query plan at the rewriting peer can be computed by neighboring peers. Hence, considering further state-of-the-art query optimization techniques developed for distributed query processing [150] in the context of PDMSs is an interesting aspect of future work.

Another sector of future work is loosely-coupled data integration. For such scenarios it is worthwhile to support automatic schema matching techniques so that peers can (semi-) automatically determine mappings. Furthermore, the network could be reorganized automatically by deriving new mappings based on existing ones, either by applying ad-hoc schema matching techniques [167] or by deriving them by mapping composition [61]. Although some interesting approaches have been developed in recent years, this remains an interesting aspect in the context of PDMSs as the benefit of reorganizing the network structure might have most beneficial influences on result quality and query execution costs.

A third sector of future work is the attempt to assign ranks or guarantees to records that are part of the result set. For example, we could rank the output to a query not only by a user-defined ranking function but also, for instance, by the distance of a record's origin to the query's initiator – the distance represents the number of times the query had to be rewritten in order to obtain the record. As mappings might be erroneous, this kind of ranking would indicate the record's correctness as with each rewriting the chance of having applied an erroneous mapping grows.

Apart from this simple assignment of a guarantee to a result record, we could alternatively consider the quality of mappings and sources. In many cases mappings are not guaranteed to be correct, they might be erroneous, incomplete, out-dated, etc. Furthermore, the data provided by a peer might be imprecise, for example due to the application of wrappers or the integration of data from multiple sources. Thus, reflecting this kind of uncertain mappings and uncertain data [52], maybe even in conjunction with the issue of trust, with a guarantee or a rank assigned to result records in a PDMS setup is an

interesting aspect of future work – especially if we consider that in a PDMS a query has to be rewritten not just once but multiple times.

Aside from mappings and source data, there is another aspect influencing result quality: the routing indexes' freshness. As routing indexes do not necessarily have to reflect the current state of the network, a query processing strategy using them might miss some relevant data to answer a query. In this context, future work might address the issue of giving freshness guarantees for the routing indexes. By combining these guarantees with result quality, we could assign each result record a rank reflecting not only data quality but also the freshness of the applied routing indexes. This kind of guarantee or rank is interesting in particular with regard to application scenarios such as disaster management, where vitally important decisions are made that should be based on the most reliable data. Maybe this guarantee is even extensible to reflect the relaxation applied by query processing strategies in order to reduce execution costs.

Applications

The literature proposes several application scenarios for PDMSs. The ones mostly referred to are: disaster management [85], mediation between ontologies in the Semantic Web [5], and information exchange between enterprises and research institutes [170]. In addition to these scenarios we see two additional ones. One of them is data exchange between astronomical observatories sharing data about their observations. Astronomical observatories not only store their data in different schemas (resulting in the kind of schema heterogeneity we have considered in this dissertation), they might also use different celestial coordinate systems. Each system uses a coordinate grid projected on the celestial sphere, in analogy to the geographic coordinate system used on the surface of the Earth. The most important coordinate systems are the equatorial coordinate system, the azimuthal coordinate system, the horizontal coordinate system, the ecliptic coordinate system, the galactic coordinate system, and the supergalactic coordinate system. As the time at which an observation is made also influences the coordinates, the observation time is another issue that must be payed attention to when exchanging data between observatories. Data integration in this context requires mappings supporting complex functions to convert coordinates from one system into another in consideration of the observation time.

In astronomy it is very common for an observatory to have multiple observations of the same astronomical object observed at different times. Moreover, the same object might be observed by multiple observatories with regard to different spectra, e.g., radio frequency, ultraviolet, x-ray, etc. Furthermore, observatories have different accuracies of measurement and might, to a small percentage, hold erroneous data. Considering all these aspects, it is difficult to decide whether two objects are the same.

Especially the fact that the amount of data is increasing every day makes it impossible to integrate the data within a single global database. In contrast, this requires a lightweight data integration scenario enabling astronomical observatories of different sizes and importance to share data – even amateur astronomers might access the system and share their observations. Note that astronomy is one of the very few sciences where amateur astronomers still play an active role. Especially in consideration of the discoveries and observations of transient phenomena, it is important to allow amateurs to share

and access data. Their data, however, might not be constantly available so that there is a certain amount of dynamic behavior the network has to deal with. As discussed above, the reliability of the sources could be reflected by assigning reliability values to the records as some kind of guarantee, which might help to identify erroneous data.

Another application scenario we see for PDMSs is data exchange between enterprises in the context of supply chains. Supply chains, or logistics networks respectively, are systems of organizations involved in manufacturing and moving a product from initial suppliers to customers. These chains typically involve multiple enterprises and organizations that might only temporarily cooperate. Although they need to exchange some information, they do not want to give away all their data, so there is only a partial mapping between the schemas of organizations.

However, future work has yet to determine to which extent the proposed extensions are realizable. In any case, we believe that the techniques combined within a PDMS may also be beneficial in a variety of applications. These applications do not necessarily have to conform to a PDMS setup but may only use some of the techniques we have discussed in this dissertation. Still, what applications these are and what techniques they will use and enhance only time will tell.

Bibliography

[1] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *PODS '98*, pages 254–263, 1998.

[2] S. Abiteboul, I. Manolescu, and E. Taropa. A Framework for Distributed XML Data Management. In *EDBT '06*, pages 1049–1058, 2006.

[3] A. Aboulnaga and S. Chaudhuri. Self-tuning Histograms: Building Histograms without Looking at Data. *SIGMOD Rec.*, 28(2):181–192, 1999.

[4] A. Aboulnaga, P. Haas, M. Kandil, S. Lightstone, G. Lohmann, V. Markl, I. Popivanov, and V. Raman. Automated Statistics Collection in DB2 UDB. In *VLDB 2004*, pages 1158–1169, 2004.

[5] P. Adjiman, P. Chatalic, F. Goasdoué, M. Rousset, and L. Simon. Distributed Reasoning in a Peer-to-Peer Setting: Application to the Semantic Web. *Journal of Artificial Intelligence Research (JAIR)*, 25:269–314, 2006.

[6] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *VLDB '96*, pages 506–521, 1996.

[7] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated Ranking of Database Query Results. In *CIDR '03*, 2003.

[8] M. Aigner. *Combinatorial Theory*, page 156 ff. Springer-Verlag Berlin, 1979.

[9] S. Amer-Yahia, S. Cho, L. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. *SIGMOD Rec.*, 30(2):497–508, 2001.

[10] W. Balke, U. Güntzer, and J. Xin Zheng. Efficient Distributed Skylining for Web Information Systems. In *EDBT '04*, pages 256–273, 2004.

[11] W. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive Distributed Top k Retrieval in Peer-to-Peer Networks. In *ICDE'05*, 2005.

[12] I. Bartolini, P. Ciaccia, and M. Patella. SaLSa: computing the skyline without scanning the whole sky. In *CIKM '06*, pages 405–414, 2006.

[13] R. Bayer. Binary B-Trees for Virtual Memory. In *ACM-SIGFIDET Workshop on Data Description, Access and Control '71*, pages 219–235, 1971.

[14] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. pages 129–139, 1988.

[15] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[16] C. Böhm, S. Berchtold, and D. Keim. Searching in High-Dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[17] A. Bonifati, E. Chang, T. Ho, and A. Lakshmanan. HePToX: Heterogeneous Peer to Peer XML Databases. In *CoRR cs.DB/0506002*, 2005.

[18] A. Bonifati, E. Chang, A. Lakshmanan, T. Ho, and R. Pottinger. HePToX: Marrying XML and heterogeneity in your P2P databases. In *VLDB '05*, pages 1267–1270. VLDB Endowment, 2005.

[19] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE '01*, pages 421–432, 2001.

[20] S. Cohen Boulakia, O. Biton, S. Cohen, Z. Ives, V. Tannen, and S. Davidson. SHARQ Guide: Finding relevant biological data and queries in a peer data management system. In *DILS '06*, 2006.

[21] I. Brunkhorst, H. Dhraief, A. Kemper, W. Nejdl, and C. Wiesner. Distributed Queries and Query Optimization in Schema-Based P2P-Systems. In *DBISP2P '03*, pages 184–199, 2003.

[22] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: a Multidimensional Workload-Aware Histogram. *SIGMOD Rec.*, 30(2):211–222, 2001.

[23] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM TODS, Vol. 27, No. 2*, 2002.

[24] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. On the Expressive Power of Data Integration Systems. In *ER '02*, pages 338–350, 2002.

[25] D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Logical foundations of peer-to-peer data integration. In *PODS '04*, pages 241–251, 2004.

[26] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Answering Regular Path Queries Using Views. In *ICDE '00*, pages 389–398, 2000.

[27] P. Cao and Z. Wang. Efficient Top-K Query Calculation in Distributed Networks. In *PODC '04*, pages 206–215, 2004.

[28] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects based on relationships. In *SIGMOD '06*, pages 371–382, 2006.

[29] C. Chan, P. Eng, and K. Tan. Efficient Processing of Skyline Queries with Partially-Ordered Domains. In *ICDE '05*, pages 190–191, 2005.

[30] C. Chan, P. Eng, and K. Tan. Stratified Computation of Skylines with Partially-Ordered Domains. In *SIGMOD '05*, pages 203–214, 2005.

[31] C. Chan, H. Jagadish, K. Tan, A. Tung, and Z. Zhang. Finding k-Dominant Skylines in High Dimensional Space. In *SIGMOD '06*, pages 503–514, 2006.

[32] C. Chan, H. Jagadish, K. Tan, A. Tung, and Z. Zhang. On High Dimensional Skylines. In *EDBT '06*, pages 478–495, 2006.

[33] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC '77*, pages 77–90, 1977.

[34] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust Cardinality and Cost Estimation for Skyline Operator. In *ICDE '06*, pages 64–74, 2006.

[35] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic Ranking of Database Query Results. In *VLDB '04*, pages 888–899, 2004.

[36] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB 1999*, pages 397–410, 1999.

[37] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing queries over multimedia repositories. *IEEE TKDE*, 16(8):992–1009, 2004.

[38] C. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. *SIGMOD Rec.*, 23(2):161–172, 1994.

[39] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE 03*, pages 717–816, 2003.

[40] S. Conrad. *Föderierte Datenbanksysteme: Konzepte der Datenintegration.* Springer-Verlag, Berlin/Heidelberg, 1997.

[41] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS '02*, pages 23–32, July 2002.

[42] F. Cuenca-Acuna, C. Peery, R. Martin, and T. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *HPDC '03*, page 236. IEEE Computer Society, 2003.

[43] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, and Y. Zhou. Parallel Distributed Processing of Constrained Skyline Queries by Filtering. In *ICDE 08*, pages 546–555, 2008.

[44] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB '96*, pages 330–341, 1996.

[45] R. de la Briandais. File Searching Using Variable Length Keys. In *Western Joint Computer Conference*, pages 295–298, 1959.

[46] E. Dellis, A. Vlachou, I. Vladimirskiy, B. Seeger, and Y. Theodoridis. Constrained subspace skyline computation. In *CIKM '06*, pages 415–424, 2006.

[47] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *PODC '87*, pages 1–12, 1987.

[48] K. Deng, X. Zhou, and H. Shen. Multi-source Skyline Query Processing in Road Networks. In *ICDE '07*, pages 796–805, 2007.

[49] Z. Despotovic and K. Aberer. P2P Reputation Management: Probabilistic Estimation vs. Social Networks. *Comput. Networks*, 50(4):485–500, 2006.

[50] H. Dhraief, A. Kemper, W. Nejdl, and C. Wiesner. Processing and Optimization of Complex Queries in Schema-Based P2P-Networks. In *DBISP2P '05*, pages 31–45, 2005.

[51] X. Dong, A. Halevy, and I. Tatarinov. Containment of Nested XML Queries. In *VLDB '04*, pages 132–143, 2004.

[52] X. Dong, A. Halevy, and C. Yu. Data Integration with Uncertainty. In *VLDB '07*, pages 687–698, 2007.

[53] C. Doulkeridis, A. Vlachou, Y. Kotidis, and M. Vazirgiannis. Peer-to-Peer Similarity Search in Metric Spaces. In *VLDB '07*, pages 986–997, 2007.

[54] O. Duschka and M. Genesereth. Query planning in infomaster. In *SAC '97*, pages 109–111, 1997.

[55] O. Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1):49–73, 2000.

[56] B. Seeger E. Dellis. Efficient Computation of Reverse Skyline Queries. In *VLDB '07*, 2007.

[57] P. Eugster, R. Guerraoui, A. Kermarrec, and L. Massoulié. From Epidemics to Distributed Computing. *IEEE Computer*, 37(5):60–67, May 2004.

[58] R. Fagin. Combining Fuzzy Information from Multiple Systems. In *PODS' 96*, pages 216–226, 1996.

[59] R. Fagin. Combining Fuzzy Information from Multiple Systems. *Journal of Computer and System Sciences*, 58(1):83–99, 1999.

[60] R. Fagin. Combining Fuzzy Information: an Overview. *SIGMOD Record*, 31(2):109–118, 2002.

[61] R. Fagin, P. Kolaitis, W. Tan, and L. Popa. Composing Schema Mappings: Second-Order Dependencies to the Rescue. In *PODS '04*, pages 83–94, 2004.

[62] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS '01*, pages 102–113, 2001.

[63] E. Franconi, G. Kuper, A. Lopatenko, and I. Zaihrayeu. Queries and Updates in the coDB Peer to Peer Database System. In *VLDB '04*, pages 1277–1280, 2004.

[64] E. Franconi, G. Kuper, A. Lopatenko, and I. Zaihrayeu. The coDB Robust Peer-to-Peer Database System. In *SEBD '04*, pages 382–393, 2004.

[65] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[66] M. Friedman, A. Levy, and T. Millstein. Navigational Plans For Data Integration. In *AAAI '99/IAAI '99*, pages 67–73, 1999.

[67] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.

[68] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *J. Intell. Inf. Syst.*, 8(2):117–132, 1997.

[69] P. Gibbons, Y. Matias, and V. Poosala. Fast Incremental Maintenance of Approximate Histograms. In *VLDB '97*, pages 466–475, 1997.

[70] GTK-Gnutella Web Page. http://gtk-gnutella.sourceforge.net/.

[71] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB '05*, pages 229–240, 2005.

[72] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *The VLDB Journal*, 16(1):5–28, 2007.

[73] L. Gong. JXTA: A Network Programming Environment. *IEEE Internet Computing*, 5(3):88–95, May/June 2001.

[74] G. Grahne and A. Mendelzon. Tableau Techniques for Querying Information Sources through Global Schemas. In *ICDT '99*, pages 332–347, 1999.

[75] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *VLDB '01*, pages 491–500, 2001.

[76] L. Gravano, P. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava. Using q-grams in a DBMS for Approximate String Processing. *IEEE Data Engineering Bulletin*, 24(2), 2001.

[77] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD '96*, pages 173–182, 1996.

[78] T. Green, G. Karvounarakis, Z. Ives, and V. Tannen. ORCHESTRA: Facilitating Collaborative Data Sharing. In *VLDB '07*, 2007.

[79] T. Green, G. Karvounarakis, N. Taylor, O. Biton, Z. Ives, and V. Tannen. ORCHESTRA: Facilitating Collaborative Data Sharing. In *SIGMOD '07 (demo)*, pages 1131–1133, 2007.

[80] D. Gunopulos, G. Kollios, V. Tsotras, and C. Domeniconi. Approximating Multi-Dimensional Aggregate Range Queries over Real Attributes. In *SIGMOD '00*, pages 463–474, 2000.

[81] U. Güntzer, W. Balke, and W. Kießling. Optimizing Multi-Feature Queries for Image Databases. In *VLDB '00*, pages 419–428, 2000.

[82] U. Güntzer, W.-T. Balke, and W. Kiessling. Optimizing multi-feature queries for image databases. In *VLDB'00*, pages 419–428, 2000.

[83] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD '84*, pages 47–57, 1984.

[84] A. Halevy. Answering Queries using Views: A Survey. *The VLDB Journal*, 10(4):270–294, 2001.

[85] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *WWW '03*, pages 556–567, 2003.

[86] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *ICDE '03*, 2003.

[87] M. Hernández and S. Stolfo. Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[88] K. Hose, A. Job, M. Karnstedt, and K. Sattler. An Extensible, Distributed Simulation Environment for Peer Data Management Systems. In *EDBT '06*, pages 1198–1202, 2006.

[89] K. Hose, M. Karnstedt, K. Sattler, and D. Zinn. Processing Top-N Queries in P2P-based Web Integration Systems with Probabilistic Guarantees. In *Proc. WebDB 2005*, pages 109–114, 2005.

[90] K. Hose, D. Klan, and K. Sattler. Distributed Data Summaries for Approximate Query Processing in PDMS. In *IDEAS '06*, pages 37–44, 2006.

[91] K. Hose, C. Lemke, J. Quasebarth, and K. Sattler. SmurfPDMS: A Platform for Query Processing in Large-Scale PDMS. In *BTW 2007*, volume 103 of *LNI*, pages 621–624. GI, 2007.

[92] K. Hose, C. Lemke, and K. Sattler. Processing Relaxed Skylines in PDMS Using Distributed Data Summaries. In *CIKM '06*, pages 425–434, 2006.

[93] K. Hose, C. Lemke, K. Sattler, and D. Zinn. A Relaxed but not Necessarily Constrained Way from the Top to the Sky. In *CoopIS 07*, pages 339–407, 2007.

[94] K. Hose, J. Quasebarth, and K. Sattler. Interoperability in Peer Data Management Systems. In *SDSI 07*, pages 152–171, 2007.

[95] K. Hose, A. Roth, A. Zeitz, K. Sattler, and F. Naumann. A Research Agenda for Query Processing in Large-Scale Peer Data Management Systems. *Information Systems Journal*, 33:597–610, 2008.

[96] V. Hristidis and Y. Papakonstantinou. Algorithms and Applications for Answering Ranked Queries Using Ranked Views. *The VLDB Journal*, 13(1):49–70, 2004.

[97] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline Queries Against Mobile Lightweight Devices in Manets. In *ICDE '06*, page 66, 2006.

[98] Z. Huang, H. Lu, B. Ooi, and A. Tung. Continuous Skyline Queries for Moving Objects. *IEEE TKDE*, 18(12):1645–1658, 2006.

[99] Y. Ioannidis. The History of Histograms (abridged). In *VLDB '03*, pages 19–30, 2003.

[100] H. Jagadish, B. Ooi, K. Tan, C. Yu, and R. Zhang. iDistance: An Adaptive B+-tree Based Indexing Method for Nearest Neighbor Search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[101] H. Jagadish, B. Ooi, and Q. Vu. BATON: a balanced tree structure for peer-to-peer networks. In *VLDB '05*, pages 661–672, 2005.

[102] S. Jeffery, M. Garofalakis, and M. Franklin. Adaptive Cleaning for RFID Data Streams. In *VLDB '06*, pages 163–174, 2006.

[103] JGraph Web Page. http://www.jgraph.com/.

[104] W. Jin, J. Han, and M. Ester. Mining Thick Skylines over Large Databases. In *PKDD '04*, pages 255–266, 2004.

[105] A. Job. Distributed Simulation of P2P Query Processing Strategies in PlanetLab (in German) – Verteilte Simulation von P2P-Anfragestrategien in PlanetLab. Master's thesis, TU Ilmenau, 2006.

[106] JXTA Web Page. https://jxta.dev.java.net/.

[107] M. Karnstedt, K. Hose, E. Stehr, and K. Sattler. Adaptive Routing Filters for Robust Query Processing in Schema-Based P2P Systems. In *IDEAS 2005, Montreal, Canada, 2005*, pages 223–228, 2005.

[108] A. Kementsietsidis and M. Arenas. Data Sharing Through Query Translation in Autonomous Sources. In *VLDB '04*, pages 468–479, 2004.

[109] A. Kementsietsidis, M. Arenas, and R. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *SIGMOD '03*, pages 325–336, 2003.

[110] M. Khalefa, M. Mokbel, and J. Levandoski. Skyline Query Processing for Incomplete Data. In *ICDE 08*, pages 556–565, 2008.

[111] W. Kießling. Preference Queries with SV-Semantics. In *COMAD '05*, pages 15–26, 2005.

[112] J. Kleinberg. Complex Networks and Decentralized Search Algorithms. In *International Congress of Mathematicians (ICM)*, 2006.

[113] A. Koch. Evaluation of Top-N Queries in P2P Systems (in German) – Auswertung von Top-N Anfragen in P2P-Systemen. Master's thesis, TU Ilmenau, 2005.

[114] V. Koltun and C. Papadimitriou. Approximately Dominating Representatives. In *ICDT '05*, pages 204–214, 2005.

[115] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[116] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB '02*, pages 275–286, 2002.

[117] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, October 1975.

[118] C. Lemke. Management of Data Summaries in PDMSs (in German) – Verwaltung von Datenzusammenfassungen in PDMS. Master's thesis, TU Ilmenau, 2007.

[119] M. Lenzerini. Data Integration: a Theoretical Perspective. In *PODS '02*, pages 233–246, 2002.

[120] U. Leser. *Query Planning in Mediator Based Information Systems*. PhD thesis, TU Berlin, Fachbereich Informatik, 2000.

[121] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[122] A. Levy, A. Mendelzon, and Y. Sagiv. Answering queries using views (extended abstract). In *PODS '95*, pages 95–104, 1995.

[123] A. Levy, A. Rajaraman, and J. Ordille. Query-Answering Algorithms for Information Agents. In *AAAI '96*, pages 40–47, 1996.

[124] A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB '96*, pages 251–262, 1996.

[125] A. Levy and D. Suciu. Deciding Containment for Queries with Complex Objects. In *PODS '97*, pages 20–31, 1997.

[126] C. Li, A. Tung, W. Jin, and M. Ester. On Dominating Your Neighborhood Profitably. In *VLDB '07*, pages 818–829, 2007.

[127] H. Li, Q. Tan, and W. Lee. Efficient progressive processing of skyline queries in peer-to-peer systems. In *InfoScale '06*, page 26, 2006.

[128] D. Liben-Nowell and J. Kleinberg. The Link Prediction Problem for Social Networks. In *CIKM 03*, 2003.

[129] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *ICDE '05*, pages 502–513, 2005.

[130] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting Stars: The k Most Representative Skyline Operator. In *ICDE '07*, pages 86–95, 2007.

[131] A. Lotem, M. Naor, and R. Fagin. Optimal aggregation algorithms for middleware. In *PODS'01*, 2001.

[132] A. Marian. Evaluating Top-k Queries over Web-Accessible Databases. In *ICDE '02*, page 369, 2002.

[133] A. Marian, N. Bruno, and L. Gravano. Evaluating Top-k Queries over Web-Accessible Databases. *ACM TODS'04*, 29(2):319–362, 2004.

[134] M. Marzolla, M. Mordacchini, and S. Orlando. Tree Vector Indexes: Efficient Range Queries for Dynamic Content on Peer-to-Peer Networks. In *PDP'06*, pages 457–464, 2006.

[135] M. Masud, I. Kiringa, and A. Kementsietsidis. Don't Mind Your Vocabulary: Data Sharing Across Heterogeneous Peers. In *CoopIS '05*, pages 292–309, 2005.

[136] S. Michel, P. Triantafillou, and G. Weikum. KLEE: a Framework for Distributed Top-k Query Algorithms. In *VLDB '05*, pages 637–648, 2005.

[137] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *PODS '02*, pages 65–76, 2002.

[138] S. Milgram. The small-world problem. *Psychology Today*, 2:60–67, 1967.

[139] D. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[140] M. Morse, J. Patel, and W. Grosky. Efficient Continuous Skyline Computation. In *ICDE '06*, page 108, 2006.

[141] K. Mouratidis, S.n Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD '06*, pages 635–646, 2006.

[142] M. Muralikrishna and D. DeWitt. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In *SIGMOD 88*, pages 28–36, 1988.

[143] N. Beckmann and H. Kriegel and R. Schneider and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD '90*, pages 322–331, 1990.

[144] Napster Web Page. http://www.napster.com.

[145] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. Edutella: a P2P Networking Infrastructure based on RDF. In *World Wide Web Conference '02*, 2002.

[146] S. Nepal and M. Ramakrishna. Query Processing Issues in Image(Multimedia) Databases. In *ICDE '99*, page 22, 1999.

[147] S. Nepal and M. Ramakrishna. Query Processing Issues in Image(Multimedia) Databases. In *ICDE '99*, page 22, 1999.

[148] ns-2 Web Page. http://www.isi.edu/nsnam/ns/.

[149] S. Oaks, B. Traversat, and L. Gong. *JXTA in a Nutshell*. O'Reilly & Associates, Inc., 2002.

[150] M. Tamer Özsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[151] P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *ICDE '03*, 2003.

[152] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD '03*, pages 467–478, 2003.

[153] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *ACM Trans. Database Syst.*, 30(1), 2005.

[154] V. Papadimos and D. Maier. Mutant Query Plans. *Information and Software Technology*, 44(4):197–206, April 2002.

[155] J. Pei, A. Fu, X. Lin, and H. Wang. Computing Compressed Multidimensional Skyline Cubes Efficiently. In *ICDE '07*, pages 96–105, 2007.

[156] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic Skylines on Uncertain Data. In *VLDB '07*, pages 15–26, 2007.

[157] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In *VLDB '05*, pages 253–264, 2005.

[158] Y. Petrakis, G. Koloniari, and E. Pitoura. On Using Histograms as Routing Indexes in Peer-to-Peer Systems. In *DBISP2P '04*, pages 16–30, 2004.

[159] Y. Petrakis and E. Pitoura. On Constructing Small Worlds in Unstructured Peer-to-Peer Systems. In *EDBT Workshops*, pages 415–424, 2004.

[160] PlanetLab Web Page. http://www.planet-lab.org/.

[161] V. Poosala and Y. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *VLDB '97*, pages 486–495, 1997.

[162] R. Pottinger and A. Halevy. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2-3):182–198, 2001.

[163] R. Pottinger and A. Levy. A Scalable Algorithm for Answering Queries Using Views. In *VLDB '00*, pages 484–495, 2000.

[164] F. P. Preparata and M. I. Shamos. *Computational Geometry - An Introduction*. Springer, 1985.

[165] J. Quasebarth. Distributed Query Rewriting in PDMSs (in German) – Verteiltes Anfrage-Rewriting in PDMS. Master's thesis, TU Ilmenau, 2007.

[166] E. Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenverarbeitung*. Addison-Wesley (Germany) GmbH, 1994.

[167] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[168] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[169] P. Rodríguez-Gianolli, A. Kementsietsidis, M. Garzetti, I. Kiringa, L. Jiang, M. Masud, R. J. Miller, and J. Mylopoulos. Data sharing in the Hyperion peer database system. In *VLDB '05*, pages 1291–1294, 2005.

[170] A. Roth and F. Naumann. System P: Query Answering in PDMS under Limited Resources. In *IIWeb '06*, 2006.

[171] A. Roth and F. Naumann. System p: Completeness-driven query answering in peer data management systems. In *BTW '07*, pages 625–628, 2007.

[172] Y. Saito and M. Shapiro. Optimistic Replication. *Computing Surveys*, 37(1):42–81, 2005.

[173] K. Sattler, S. Conrad, and G. Saake. Interactive Example-driven Integration and Reconciliation for Accessing Database Federations. *Information Systems*, 28(5):393–414, 2003.

[174] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. HyperCuP – Hypercubes, Ontologies and Efficient Search on P2P Networks. In *International Workshop on Agents and Peer-to-Peer Computing*, pages 112–124, 2002.

[175] H. Shang and T. Merrettal. Tries for Approximate String Matching. *IEEE Trans. on Knowl. and Data Eng.*, 8(4):540–547, 1996.

[176] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *VLDB '06*, pages 751–762, 2006.

[177] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[178] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. ISOMER: Consistent Histogram Construction Using Query Feedback. In *ICDE '06*, page 39, 2006.

[179] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *VLDB '01*, pages 19–28, 2001.

[180] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01*, pages 149–160, 2001.

[181] E. Sutinen and J. Tarhio. On Using q-Gram Locations in Approximate String Matching. In *ESA '95*, pages 327–340, 1995.

[182] E. Sutinen and J. Tarhio. Filtration with q-Samples in Approximate String Matching. In *CPM '96*, pages 50–63, 1996.

[183] K.-L. Tan, P.-K. Eng, and B. Chin Ooi. Efficient progressive skyline computation. In *Proceedings of VLDB*, pages 301–310, 2001.

[184] Y. Tao and D. Papadias. Maintaining Sliding Window Skylines on Data Streams. *IEEE TKDE*, 18(3):377–391, 2006.

[185] Y. Tao, X. Xiao, and J. Pei. SUBSKY: Efficient Computation of Skylines in Subspaces. In *ICDE '06*, page 65, 2006.

[186] Y. Tao, X. Xiao, and J. Pei. Efficient Skyline and Top-k Retrieval in Subspaces. *IEEE TKDE*, 19(8):1072–1088, 2007.

[187] I. Tatarinov and A. Halevy. Efficient query reformulation in peer data management systems. In *SIGMOD '04*, pages 539–550, 2004.

[188] I. Tatarinov, Z. Ives, J. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The Piazza Peer Data Management Project. *SIGMOD Record*, 32(3):47–52, 2003.

[189] N. Taylor and Z. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD '06*, pages 13–24, 2006.

[190] M. Theobald, G. Weikum, and R. Schenkel. Top-k Query Evaluation with Probabilistic Guarantees. In *VLDB'04*, pages 648–659, 2004.

[191] L. Tian, L. Wang, A. Li, P. Zou, and Y. Jia. Continuous Skyline Tracking on Update Data Streams. In *APWeb/WAIM Workshops*, pages 192–197, 2007.

[192] E. Ukkonen. Approximate String Matching with q-grams and Maximal Matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[193] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Rockville, MD., USA, 1989.

[194] J. Ullman. Information Integration Using Logical Views. In *ICDT '97*, pages 19–40, 1997.

[195] J. Ullmann. A Binary n-Gram Technique for Automatic Correction of Substitution, Deletion, Insertion and Reversal Errors in Words. *The Computer Journal*, 20(2):141–147, 1977.

[196] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis. SKYPEER: Efficient Subspace Skyline Computation over Distributed Data. In *ICDE '07*, pages 416–425, 2007. To appear.

[197] A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis. Skyline-based Peer-to-Peer Top-k Query Processing. In *ICDE*, pages 1421–1423, 2008.

[198] S. Wang, B. Ooi, A. Tung, and L. Xu. Efficient Skyline Query Processing on Peer-to-Peer Networks. In *ICDE '07*, pages 1126–1135, 2007.

[199] M. Weis and F. Naumann. DogmatiX Tracks down Duplicates in XML. In *SIGMOD '05*, pages 431–442, 2005.

[200] G. Wills. NicheWorks - Interactive Visualization of Very Large Graphs. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 403–414, 1997.

[201] R. Wong, A. Fu, J. Pei, Y. Ho, T. Wong, and Y. Liu. Efficient Skyline Querying with Variable User Preferences on Nominal Attributes. *CoRR*, abs/0710.2604, 2007.

[202] P. Wu, D. Agrawal, Ö. Eğecioğlu, and A. Abbadi. DeltaSky: Optimal Maintenance of Skyline Deletions without Exclusive Dominance Region Generation. In *ICDE '07*, pages 486–495, 2007.

[203] P. Wu, C. Zhang, Y. Feng, B. Zhao, D. Agrawal, and A. Abbadi. Parallelizing Skyline Queries for Scalable Distribution. In *EDBT '06*, pages 112–130, 2006.

[204] T. Xia and D. Zhang. Refreshing the sky: the compressed skycube with efficient support for frequent updates. In *SIGMOD '06*, pages 491–502, 2006.

[205] J. Xin, G. Wang, L. Chen, X. Zhang, and Z. Wang. Continuously Maintaining Sliding Window Skylines in a Sensor Network. In *DASFAA 07*, pages 509–521, 2007.

[206] L. Yan and M. Özsu. Conflict Tolerant Queries in AURORA. In *CoopIS 99*, pages 279–290, 1999.

[207] K. Yee, D. Fisher, R. Dhamija, and M. Hearst. Animated Exploration of Dynamic Graphs with Radial Layout. In *INFOVIS '01*, page 43, 2001.

[208] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient Maintenance of Materialized Top-k Views. In *ICDE '03*, pages 189–200, 2003.

[209] C. Yu, B. Ooi, K. Tan, and H. Jagadish. Indexing the Distance: An Efficient Method to KNN Processing. In *VLDB '01*, pages 421–430, 2001.

[210] C. Yu and L. Popa. Semantic Adaptation of Schema Mappings when Schemas Evolve. In *VLDB '05*, pages 1006–1017, 2005.

[211] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. Xu Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB '05*, pages 241–252, 2005.

[212] L. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1969.

[213] D. Zinn. Skyline Queries in P2P Systems. Master's thesis, TU Ilmenau, 2005.