Query Processing in a DHT-Based Universal Storage

The World as a Peer-to-Peer Database

Marcel Karnstedt

Dissertation

zur Erlangung des akademischen Grades / in partial fulfillment of the requirements for the degree of

Doktoringenieur (Dr.-Ing.)

vorgelegt der / submitted to

Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

–

Faculty of Computer Science and Automation
Ilmenau University of Technology

von / by

Dipl.-Inf.

geboren am / born on

29.12.1977 in Halle (Saale)

vorgelegt am / submitted on

24.02.2009

Wissenschaftliche Aussprache am / oral examination on

18.08.2009

Gutachter / Reading Committee

1. Prof. Dr. habil. Kai-Uwe Sattler
2. Prof. Dr. Manfred Hauswirth
3. Prof. Alfons Kemper, Ph.D.

Dedicated to Anja

Zusammenfassung

Mit dem Aufschwung des Internets wachsen Anwendungen immer weiter, sowohl bezüglich des Datenvolumens als auch der Anzahl von Teilnehmern. Gleichzeitig wächst die Größe und räumliche Ausdehnung der unterliegenden Netzwerkstrukturen. Viele dieser Anwendungen bieten Zugriff auf riesige Datenmengen für eine ebenso riesige Anzahl von Teilnehmern. Wenn es sich um Daten von öffentlichem Interesse handelt die der Öffentlichkeit zur Verfügung stehen, werden solche Anwendungen als Public Data Management klassifiziert. Oftmals sind die Daten auch offen für Erweiterung und Veränderung. Folglich entstehen die Nutzdaten durch Kollaboration und werden von der Gemeinschaft gepflegt.

Strukturierte Daten und entsprechende Zugriffsmethoden sind dabei oft eine zwingende Voraussetzung dieser Systeme. Dabei ist ein hoher Grad an Heterogeneität zu erwarten. Funktionalität aus traditionellen Datenbank- und Integrationssystemen ist oft gefragt, erwartet, oder sogar benötigt um solche Datenmengen sinnvoll zu handhaben. Zentralisierte Ansätze sind beschränkt in ihrer Skalierbarkeit und können mit den stetig wachsenden Anforderungen nicht Schritt halten. Die Verteilung von Daten und Last ist eine logische und notwendige Konsequenz. Verteilung in kleinem Maßstab, so wie in Clusters und lokal abgeschotteten Systemen, stößt ebenso an seine Grenzen. Stetige Investitionen sind nicht möglich, denn oft findet sich kein einzelner Nutzer und keine Instanz, die bereit und befähigt sind, die enormen Kosten solcher Systeme zu tragen. Im Gegenzug resultieren Public Data Management Anwendungen aber auch in verringerten Anforderungen, wie zum Beispiel gelockerten Garantien bezüglich Konsistenz und Verfügbarkeit. Das macht aus der Idee der totalen Verteilung von Daten und Last einen vielversprechenden und praktikablen Ansatz. Auch wenn existierende Systeme bereits entsprechende Funktionalität bieten, so ist diese in der Regel stark vereinfacht und extrem eingeschränkt. Ein Großteil der Anforderungen von datenbank-ähnlichen verteilten Systemen wird weder unterstützt noch ermöglicht.

Diese Arbeit konzentriert sich auf die Anforderungen und ungelösten Herausforderungen die speziell im Kontext der Anfrageverarbeitung auftreten. Neben der reinen Effizienz der Anfrageverarbeitung umfasst dies Punkte wie die Ausdruckskraft von Anfragen und die Anwendbarkeit bestimmter Konzepte. Wir erfassen zunächst diese und weitere Herausforderungen und motivieren den Beitrag dieser Arbeit. Im Anschluss schlagen wir entsprechende Techniken und Ansätze für effiziente Anfrageverarbeitung in weit-verteilten Systemen vor. Wir empfehlen eine Schichtenarchitektur, welche die Vorteile dezentralisierter Peer-to-Peer-Ansätze nutzt. Basierend auf dieser Architektur, einem vertikalen Tripelmodell und einer zugehörigen Anfragealgebra, stellen wir eine physische Algebra vor. Die Operatoren dieser Algebra nutzen Parallelität und andere Paradigmen der verteilten Anfrageverarbeitung. Des weiteren präsentieren wir Techniken um komplexe Anfragepläne die Nutzeranfragen repräsentieren zu verarbeiten. Das Anfragesystem ist erweitert um ein Kostenmodell, welches eine sinnvolle Planung von Anfragen ermöglicht. Wir diskutieren ebenfalls Garantien und Vorhersagbarkeit. In diesem Zusammenhang stellen wir einen leichtgewichtigen und flexiblen Ansatz zur Schätzung der Vollständigkeit von Teilantworten vor. Eine umfassende Evaluierung auf der Basis einer Referenzimplementierung stellt die Anwendbarkeit und Korrektheit der vorgestellten Konzepte unter Beweis. Dies unterstreicht den wertvollen Beitrag zur Entwicklung riesiger verteilter Datensysteme.

Abstract

With the growth of the Internet, novel applications scale significantly in both, data and the number of users. In parallel, the size and local expansion of underlying network infrastructures increase as well. A wide range of such applications provides access to huge data collections to a likewise huge number of participants. If these data are of public interest and open to the public, according applications are classified as Public Data Management. Often, data are also open to extension and modification. Thus, data get into existence through collaboration and are maintained by the community.

Structured data and according access methods are often a mandatory requirement to make these systems work. In doing so, a high degree of heterogeneity is very probable. Features known from traditional database and data integration systems are often wanted, expected, or even required to handle such data collections meaningfully. The applicability of centralized approaches is limited in scale and cannot keep with the ever-growing requirements. Distribution of data and load is an intuitive and mandatory consequence. Distribution in small scale, such as in clusters and locally encapsulated systems, gets to its limits as well. The "kill it with iron" strategy is not applicable, because often no instance or single user is willed and enabled to bear the immense costs of resulting systems. In turn, Public Data Management applications also result in lowered requirements, such as relaxed guarantees in consistency and availability. This turns the idea of a total distribution of all data and load into a promising and feasible approach. Even if existing systems already provide according features, these are in general by far simplified and strongly limited. A wide range of requirements of database-like distributed systems is still neither supported nor enabled.

This work focuses on the requirements and unsolved challenges that occur specifically in the context of query processing. Besides the pure efficiency of query processing, this also includes issues like query expressiveness and practicability of certain concepts. We first identify these and other challenges and motivate the contributions of this work. Afterwards, we propose according techniques and approaches for efficient query processing in widely distributed systems. We suggest a layered architecture that makes use of the advantages of decentralized Peer-to-Peer approaches. Based on this, a vertical triple model, and an according query algebra, we propose a physical query algebra. This algebra comprises operators that make excessive use of parallelism and other paradigms of distributed query processing. Further, we present techniques for processing complex query plans that represent user queries. The query framework is extended by an appropriate cost model to enable meaningful query planning. We also discuss guarantees and predictability. In this context, we introduce a light-weight and flexible approach for estimating the completeness of partly query answers. An extensive evaluation based on a reference implementation shows the applicability and correctness of the proposed concepts. This highlights the valuable contribution for the development of large-scale distributed data systems.

Acknowledgements

First of all, I want to thank Kai-Uwe Sattler. He picked me up in Halle and introduced me to the exciting world of research. Without his help and the things I learned, this thesis would have never come into existence. Also a lot of thanks to Manfred Hauswirth, who supported and motivated me by providing interesting facts and insights into research and the accompanying techniques. Further, I want to thank all the colleagues for the valuable discussions and good times we always had. This particularly concerns Katja Hose, who was the first student I worked with and became one of the best colleagues, and Daniel Klan, my pleasant and appreciated officemate. Special thanks to Thorsten Strufe for the discussions and the enduring good mood I was in when I walked down the floor – and for the motivation he provided with his successful graduation. Further thanks to the students who supported me in implementation issues and contributed with valuable ideas during their Diploma theses. This includes in the first line Martin Richtarsky, Michael Haß, and Jessica Müller. Cheers to Ioana for the big help in preparing the final book, tu eşti cea mai bună. Gigantic thanks to all my friends and my family! My parents always supported me in each conceivable context, thanks to mom and dad! Thanks to Anja, the woman I was loving over all, for the masses of energy, power, and confirmation. And yes, also thanks for your nagging. In the end, it was mostly helpful. I will never forget the great time we had, and I am sorry for the tough moments I gave you. My love will never totally fade away. Thanks to my closest friends, my brother Steven, Christian, Dirk, Uli, Volker, and my almost brother Brodi – you always gave me the feeling that I can and will achieve whatever I plan. I want to thank all of you and all those I did not mention explicitly for the motivation and encouragement, the good times, and for confirming the quality of my work!

Contents

Zusammenfassung v
Abstract vii
Acknowledgements ix
List of Figures xiv
List of Algorithms xviii
List of Tables xx
List of Symbols xxii

I The Problem 1

1 Motivation 3
1.1 Introduction 3
1.2 Public Data Management 5
1.2.1 Application Scenarios 6

2 Problem Statement 11
2.1 Challenges 11
2.2 Contribution and Structure 15

II Large-Scale Distributed Data Management 19

3 Architectural Approaches 21
3.1 State of the Art 21
3.2 The P2P Paradigm 26
3.2.1 Unstructured P2P Systems 28
3.2.2 Structured P2P Systems 29
3.2.3 Super-Peer Architectures 31
3.3 A Layered Architecture 32

4 Distribution Layer 35
4.1 DHTs in the Distribution Layer 36
4.1.1 The P-Grid P2P System 37
4.2 Data Model 41
4.2.1 Indexing Relational User Data 41
4.2.2 Indexing XML User Data 44
4.2.3 Representing Semantic Correspondences 46

4.3 Related Work . . . 48

III Query Layer **51**

5 Query Language **53**
5.1 Vertical Query Language and Algebra . . . 53
5.1.1 Using Views for Representing Mappings . . . 76
5.1.2 Integrating Semantic Correspondences . . . 77
5.2 Related Work . . . 84

6 Physical Query Operators and Processing Strategies **87**
6.1 Basic Query Operators . . . 88
6.2 Advanced Query Operators . . . 97
6.2.1 Similarity Operators . . . 98
6.2.2 Ranking Operators . . . 106
6.3 Other Operators . . . 119
6.4 Processing Query Plans . . . 123
6.4.1 Pipelining . . . 125
6.4.2 Overview . . . 127
6.4.3 Query Planning and Adaptive Query Processing . . . 128
6.5 Costs of Physical Operators . . . 130
6.6 Related Work . . . 138

7 Guarantees and Predictability **147**
7.1 Query Completeness in Data Systems . . . 148
7.2 Estimating Query Completeness on Peer Level . . . 149
7.2.1 Overlay-Intern Completeness Estimation . . . 159
7.3 Guarantees . . . 163
7.4 Related Work . . . 170

IV Evaluation and Conclusion **173**

8 Implementation and Evaluation **175**
8.1 UniStore: A Light-Weight Internet-Scale Universal Storage . . . 175
8.1.1 Overview of Implementation . . . 175
8.1.2 Special Implementation Issues . . . 178
8.2 Evaluation . . . 179
8.2.1 Experimental Setup . . . 179
8.2.2 Metrics . . . 184
8.2.3 Physical Operators and Processing Strategies . . . 186
8.2.4 Further Issues of Query Plan Processing . . . 202
8.2.5 Query Completeness . . . 205
8.2.6 Integrating Semantics . . . 216
8.2.7 Summary . . . 218

9 Conclusion and Outlook **221**
9.1 Summary 221
9.2 Open Issues and Future Directions 222

A Basic VQL Syntax **225**

List of Figures

3.1 Classification of distributed systems . . . 22
3.2 Example topologies in structured P2P overlays . . . 30
3.3 Layered architecture of a large-scale universal storage . . . 32

4.1 P-Grid overlay network . . . 39
4.2 Range query illustration . . . 40
4.3 Triples in the P-Grid overlay network . . . 44
4.4 Example schema correspondences . . . 46

5.1 Example schema . . . 68
5.2 Unexpanded query . . . 81
5.3 Query with mappings for expansion . . . 81
5.4 Expanded query . . . 82
5.5 Expansion using ontology . . . 83

6.1 Routing schemes for $\xi_{['A']}$ . . . 92
6.2 Routing schemes for $\omega^{HASH}_{?r;['A']}$ . . . 95
6.3 Similarity materialization on different levels . . . 103
6.4 Routing schemes for $\bowtie^{HASH}$ with $d = 2$ . . . 105
6.5 Principle of φ . . . 106
6.6 Frame skyline approach . . . 114
6.7 Projections in a three-dimensional skyline . . . 117
6.8 Narrowed search space in a three-dimensional skyline . . . 118
6.9 Principle of Φ^{PAR} . . . 119
6.10 Example query expansion . . . 121
6.11 Example query plan . . . 123
6.12 Different combinations of ξ operators ($|A| := |p_{r_A}|, |B| := |p_{r_B}|$) . . . 124
6.13 Routing scheme for $\xi^{SEQ}_{['A']}$ with peer pipelining . . . 126
6.14 Example of utilizing mutations for adaptive query processing . . . 129
6.15 Late vs. early synchronization . . . 130
6.16 Example plans for cost estimation . . . 136

7.1 Result quality in Planet-Lab . . . 149
7.2 Sequential processing . . . 151
7.3 Intra-operator parallel processing . . . 151
7.4 Example query . . . 156
7.5 Completeness for parallel join . . . 158

7.6 Completeness for parallel aggregate . . . 158
7.7 Estimating the P-Grid trie . . . 161
7.8 Minimal routing trees for complete DOR . . . 165
7.9 Minimal routing trees for complete FNR . . . 166
7.10 FNR example for minimal, maximal, and average estimation technique . . . 166
7.11 DOR estimation after two replies from prefix 01 . . . 168

8.1 Implementation overview . . . 176
8.2 GUI screenshot . . . 177
8.3 Q-gram storage for DBPedia and Mondial data . . . 181
8.4 Key distribution . . . 182
8.5 Shape of evaluation queries and class identifiers . . . 182
8.6 Shape of queries for semantic expansion . . . 184
8.7 Comparison of ξ^{RQ} and ξ^{SEQ} for two different VTPs . . . 186
8.8 Cost comparison of ξ^{RQ} and ξ^{SEQ} . . . 187
8.9 ξ^{RQ} vs. ξ^{SEQ} on small ranges . . . 188
8.10 Range- and hash-based extractions vs. lookups . . . 188
8.11 ω^{RQ} vs. $\omega^{HASH,SEQ}$ for different numbers of materializations . . . 189
8.12 Cost comparison of ω variants . . . 190
8.13 Variants of ω for different numbers of materializations . . . 191
8.14 First reply times for different numbers of materializations . . . 192
8.15 Combination of ξ and ω . . . 192
8.16 $\bowtie^{LOC}$ vs. $\bowtie^{SYNC}$ for different numbers of materialized attributes . . . 193
8.17 Similarity extraction . . . 194
8.18 First reply time for similarity extraction . . . 195
8.19 Similarity extraction for different distances . . . 196
8.20 Cost comparison of similarity extractions . . . 197
8.21 Unbound extraction for similarity on schema level . . . 197
8.22 Similarity on schema level . . . 198
8.23 Schema similarity for different distances . . . 198
8.24 Similarity join operators . . . 199
8.25 Reply count and robustness for similarity joins . . . 200
8.26 Schema similarity joins with unbound extraction . . . 200
8.27 Effect of different limits for φ^{SYNC} . . . 201
8.28 Tuple pipelining . . . 201
8.29 25 peers . . . 202
8.30 50 peers . . . 203
8.31 74 peers . . . 203
8.32 Messages . . . 204
8.33 Full-scale Planet-Lab experiments . . . 204
8.34 Q-gram queries with 384 Planet-Lab peers . . . 205
8.35 Replication:1, maxref:1 . . . 206
8.36 Replication:1, maxref:3 . . . 206
8.37 Replication:1, maxref:5 . . . 207
8.38 Replication:2, maxref:1 . . . 207
8.39 Replication:2, maxref:3 . . . 208

8.40 Replication:2, maxref:5 . . . 208
8.41 Estimated relative number of replies with corrections . . . 209
8.42 Accuracy for separated routing methods . . . 210
8.43 Sequential operators . . . 211
8.44 Minimal, maximal, and average estimation for s_2 . . . 211
8.45 Minimal, maximal, and average estimation for c . . . 212
8.46 Minimal, maximal, and average estimation for a and ac . . . 212
8.47 Impact of MiMes . . . 213
8.48 Extra effort for MiMes . . . 213
8.49 Queries on Planet-Lab . . . 214
8.50 Errors in c on 393 peers . . . 214
8.51 Planet-Lab queries on Imdb data . . . 215
8.52 Aggregation query with the use of completeness estimation . . . 215
8.53 Query expansion . . . 216
8.54 Overhead . . . 217
8.55 Performance under different variations . . . 217

List of Algorithms

1 General processing of $\xi^{SEQ}_{[\texttt{'A'}]}$ . . . 92
2 General processing of $\xi^{RQ}_{[\texttt{'A'}]}$ . . . 93
3 General processing of $\omega^{HASH,PAR}_{?r;[\texttt{'A'}]}$ . . . 94
4 General processing of $\bowtie^{HASH,PAR}_{?A=?B}(\xi_{[\texttt{'B'}]})$ . . . 96
5 Q-gram-based similarity extraction: $\sigma^{QG,PAR}_{edist(?A,s)\leq d}(\xi_{[\texttt{'A'}]})$ . . . 103
6 Q-gram-based similarity join: $\bowtie^{HASH,PAR}_{edist(?A,?B)\leq d}(\xi^{QG,PAR}_{[\texttt{'B'}]})$ . . . 104
7 Top-N query: $\varphi^{PAR}_{\texttt{NN},m,0;?A;N}(\xi_{[\texttt{'A'}]})$ – initial phase . . . 107
8 Collection phase top-N-collect() used in $\varphi^{PAR}_{\texttt{NN},m,0;?A;N}(\xi_{[\texttt{'A'}]})$ . . . 108
9 Synchronization phase top-N-sync() used in $\varphi^{PAR}_{\texttt{NN},m,0;?A;N}(\xi_{[\texttt{'A'}]})$ . . . 109
10 Top-N query on strings: $\varphi^{PAR}_{\texttt{NN},m,0;?A;N}(\xi_{[\texttt{'A'}]})$ – initial phase with sketched synchronization phase . . . 112
11 Initialization for $\Phi_{\texttt{MIN},\ldots,\texttt{MIN};?A_1,\ldots,?A_n}(\omega_{?A_i;[\texttt{A}_1],\ldots,[\texttt{A}_{i-1}],[\texttt{A}_{i+1}],\ldots,[\texttt{A}_n]}(\xi_{[\texttt{A}_i]}))$ . . . 114
12 `query-frame()` for $\Phi_{\texttt{MIN},\ldots,\texttt{MIN};?A_1,\ldots,?A_n}(\omega_{?A_i;[\texttt{A}_1],\ldots,[\texttt{A}_{i-1}],[\texttt{A}_{i+1}],\ldots,[\texttt{A}_n]}(\xi_{[\texttt{A}_i]}))$ . . . 115
13 `det-candidates()` for $\Phi_{\texttt{MIN},\ldots,\texttt{MIN};?A_1,\ldots,?A_n}(\omega_{?A_i;[\texttt{A}_1],\ldots,[\texttt{A}_{i-1}],[\texttt{A}_{i+1}],\ldots,[\texttt{A}_n]}(\xi_{[\texttt{A}_i]}))$ . . . 116
14 `finish()` for $\Phi_{\texttt{MIN},\ldots,\texttt{MIN};?A_1,\ldots,?A_n}(\omega_{?A_i;[\texttt{A}_1],\ldots,[\texttt{A}_{i-1}],[\texttt{A}_{i+1}],\ldots,[\texttt{A}_n]}(\xi_{[\texttt{A}_i]}))$ . . . 118
15 ω_P and $\xi^{SEQ}_{[\texttt{'A'}]}$ combined by triple set pipelining . . . 125
16 General plan processing `process(`Q`)` . . . 127
17 Basic algorithm run with each received reply . . . 153

List of Tables

4.1 DHT procedure calls used in this work 37

5.1 Logical query plan operators 69

6.1 Symbols used in cost formulas 131
6.2 Asymptotical cost approximations 132
6.3 Costs of physical operators 132
6.4 Example statistics needed for cost estimation 135
6.5 Cost estimation for Q_1 ($rev_\varphi = 0.1$) 136
6.6 Cost estimation for Q_1 with inter-operator parallel blocking join ($rev_\bowtie = 0, rev_\varphi = 0.1$) 137
6.7 Cost estimation for Q_2 ($rev_\varphi = 0$) 137

List of Symbols

a	Amount of result data already received
α	Final amount of expected result data
r	Number of replies already received
ρ	Final Number of replies expected
R	Number of final replies currently estimated
ot	Originally inserted tuple
OT	Set of triples injected
ts	Result triple set
RT	Set of triple sets of a query's result
t	Triple
	Triple part delimiter
$\cdot$	Triple part delimiter in query plans
T	Set of triple sets as operator input
oid	Object ID
a	Attribute name
v	Attribute value
h	Hash function
k	Hash key
$\|\|$	Concatenation of strings
Π	Key that a peer is responsible for
I	Number of indexes built
ref	Part of a triple a variable refers to
LOC	Local implementation of an operator
SEQ	Sequential implementation of an operator
PAR	Parallel implementation of an operator
RQ	Range-query-based implementation of an operator
HYB	Hybrid implementation of an operator
$SYNC$	Implementation of an operator that needs synchronization
$HASH$	Hash-based implementation of an operator
QG	Q-gram-based implementation of an operator
qg	Symbol used for a q-gram
#	Leading sign in q-grams
‡	Trailing sign in q-grams
c_{rq}	Rounds needed in sophisticated ranking operators
$\lceil\lceil$	Left sign for maximal relation
$\rceil\rceil$	Right sign for maximal relation

$\lfloor\lfloor$	Left sign for minimal relation
$\rfloor\rfloor$	Right sign for minimal relation
$edist$	Edit distance function
e	Class of evaluation queries: extraction
m	Class of evaluation queries: materialization
c	Class of evaluation queries: complex plan
a	Class of evaluation queries: aggregation
ac	Class of evaluation queries: complex aggregation
s	Class of evaluation queries: queries with mapping parts
ξ	Extraction operator
ω	Materialization operator
σ	Selection operator
π	Projection operator
$\times$	Cross-product operator
$\bowtie$	Join operator
φ	Ranking operator
Φ	Skyline operator
μ	Map operator
γ	Grouping operator

Part I

The Problem

Chapter 1

Motivation

This chapter provides an overview of the problems that this work focuses on and the application scenarios it considers. After a general introduction in Section 1.1, we introduce example applications in the context of Public Data Management in Section 1.2. These sections highlight the open issues of these applications and motivate the practical needs for the contributions of this work.

1.1 Introduction

Recent years have been embossed by processes of continuous growth, expansion, and union. This has involved companies, organizations, lobbies, stakeholders, and more. But, not only markets and their players steadily grow and get united with ongoing globalization. Also, communities scale to millions, data repositories get larger and larger, data sources get united, and people start to explore the power and possibilities of sharing data and knowledge. This development spans all areas of society. Computer information systems play a special role in this game: Their rapid development over the last decades is a perfect example of a steady expansion. On the other side, it is a fundamental prerequisite as well as motor facilitating the same development in other areas of society. Expansion and union of computer systems are recently driven by means of extended collaboration. In first line, this is exemplified by the amazing success story of the Internet. But, also other outcomes of recent developments based on ideas of global consolidation and collaboration affirm these trends. This started from (meanwhile) relatively small cluster systems and distributed database systems. It evolved into approaches like the Grid [69], the Peer-to-Peer (P2P) [8] paradigm, and novel phenomena like folksonomies and friend-of-a-friend networks [68], pedias, and other Semantic X approaches combined under the flag of Web 2.0 [1].

For researchers in the field of database and information systems this reveals a huge variety of challenges as well as opportunities. We are moving from a world of separated data silos to a world of large-scale information systems, which offer access to all-embracing data almost everywhere and anytime. Ubiquitous access to data of any kind is a main requirement of a large fraction of current developments in Information Technology (IT). The idea of sharing resources (processing power, knowledge, storage, etc.) between mil-

[1] http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

lions of users gets more important every day. Concerning information systems, a clear trend shows that data repositories and collections of knowledge are more and more created by collaborating communities. This includes groups of arbitrarily related people, institutions, sites, etc. The idea is to see "The world as a database". There is a wealth of applications where this idea needs to be backed by distributed information systems that provide efficient access to data distributed at large scale. As an example, consider *Public Data Management*, where data of public interest is maintained by the community and accessed by the community. Preferably, costs of the infrastructure are shared as well. Search should be issued from a local point of view. That is, without any need for special or global knowledge, even in very heterogeneous environments. We give an informal definition of Public Data Management and introduce actual application scenarios in Section 1.2.

Although traditional database systems are originally designed for centralized repositories, adopting cognitions and experiences from this well-investigated field of research is mandatory for successfully supporting and enabling the described development. But, in their original form most approaches and techniques do not scale enough to achieve this. A main problem is the inappropriateness of the underlying closed world assumption. This means, as long as data are not updated, the same query will always return identical results, which are always complete and always available. In systems revealing their strengths from the introduced large-scale collaboration of loosely-coupled users this inherently needs to be replaced by the open world assumption. As not all participants are online all time, data may only be available periodically. Due to times of high load or technical issues, fractions of networks may be down or not reachable. Thus, the hosted data are not available as well. Even during processing a query, hosts can fail, leave or join the network, or simply send no or delayed answers [78]. Similar, other fundamentals and assumptions that traditional database systems rely on need to be relaxed or even abandoned at all. This results in the need for analyzing, discussing, and rethinking major paradigms and methods. It spans from system design over data models and query processing techniques to problems of trust and reputation.

This work focuses on the specific challenges occurring in information sharing, searching, and filtering in huge loosely-coupled data systems. Search and filtering is expressed in terms of queries. We investigate query processing from a database point of view and discuss how to port it to large-scale distributed storage systems. We introduce a chosen set of representative application scenarios in Section 1.2. Afterwards, we discuss the specific challenges arising from these applications in Section 2.1. Section 2.2 clearly delimits the investigated problems and lists the contributions of the work in hand.

In Part II, we suggest a distributed data management and storage scheme as a basis for further considerations. Based on this, we focus on requirements, techniques, solutions, and open issues in the context of query processing in Part III of this work. We implemented most of the discussed concepts and approaches in a prototype system, which is introduced in Part IV. This implementation is used for an extensive evaluation. Finally, Part IV sums up the contributions, open issues, and further directions resulting from this work.

1.2 Public Data Management

In this section, we introduce several selected application scenarios. These scenarios have in common that they are based on the idea of collaboration and exchange between a large number of participants. All the applications benefit from a distributed storage system as a basic ingredient. Although the exact requirements may differ in detail, they all result in identical or similar challenges. Before discussing these challenges in Section 2.1, we exemplify them through the practical situations in which they occur. Specifically, we refer to issues concerning distributed management and querying of large data collections.

As mentioned before, there is an evolving trend that computer users create and maintain large data collections of public interest through collaboration. Each user may individually provide parts of the data and gains (theoretically unlimited) access to that global collection. The information, their structure, and their semantics are controlled by the users. Data management, data integration, and data filtering come into existence through the collaboration of a large number of participants. Information is of general and public interest. Early examples are popular file sharing systems like Napster [144] and Gnutella [74]. Meanwhile, the idea gained ground in many other collaborative systems: knowledge bases like Wikipedia [211], recommender systems like Ciao.com [51], city and travel guides [205], social networks [143], and so on. In these systems the power of many shows up. Due to the rapid growth of the Internet, most of the mentioned systems meanwhile scale up to millions of users. Following, the available data scales as well. So do the numbers of different user groups, interests, communities, and attending generations of people.

Still, many of these systems are based on a centralized infrastructure. Recent trends show that, besides privacy issues, centralized approaches do not scale enough to meet the ever-growing requirements of the modern IT society. Rather, the costs for providing the infrastructure should be shared by the users in a fair manner. If the total costs of ownership can be shared among nodes at the edge of the network infrastructure, i.e., user machines, why should specialized high-end servers, expensive clusters, or costly centralized database software be used? Moreover, in order to bridge the gap from separated information silos to truly valuable knowledge bases, data should be shared between different systems as well. Decentralization allows to share resources for managing and searching the resulting large data sets. Data should be distributed among the participants and search should come into effect by collaboratively processing queries in a distributed manner. This is not restricted to distributing only raw user data. Rather, users often tend to wish to control their data, location of data, and maybe accessibility. But, in order to search and locate data efficiently, distributed indexes at large scale are a very powerful and promising tool. In this context, querying should support to get (interactively refined) overviews over the data and still guarantee low response times. A modified understanding of actuality and completeness is needed due to the applied open world assumption. These measures have to be represented accordingly.

In such systems, data are usually structured in some way. Gathering all data in a centralized repository and trying to impose participants to use a global schema is almost impossible. Rather, each participant may use an individual structure, such as an own relational schema. Others may prefer to use XML [216] or other semi-structured formats for exchanging data. Even when sharing binary objects like videos, music files,

pictures, etc. data are usually annotated by some semantic metainformation. Metadata and raw data are structured conforming to the individual needs and characteristics (like nationality and language) of their provider. But, although these huge data collections tend to be very heterogeneous, users expect to be enabled to query solely from their local point of view. Nevertheless, they aim for accessing all available data, even if the exact structure is unknown to them.

It is not only heterogeneity in the different user schemata that demands for exceptionally generic and flexible data models. The characteristics of modern Web applications require a rethinking in the way schemata are handled. In Public Data Management scenarios users join communities in an ad-hoc fashion in order to collaborate, discuss, and curate online. Topics and interests, and thus, data, change with time. As the users will rarely be able to agree on data schemata ahead of time, structure of data will be very dynamic. Schemata will still be needed, but should be discoverable from the data itself. Though, these systems need to be easy to use. This demands techniques for automatically handling heterogeneities and dynamics in data and schemata.

1.2.1 Application Scenarios

Public Data Management, as introduced above, is a kind of major paradigm based on the idea of collaboration. There is a multitude of concrete application scenarios that evolved from this paradigm. In the following, we introduce four specifically popular applications, which all apply facets of Public Data Management up to a certain degree. This section helps to identify the common challenges that we have to face in order to make these applications work meaningfully and efficiently.

Scientific Data Collections and Computing Confirming to [6], biological knowledge currently publicly available is distributed over 700 separated databases. Each of these databases potentially has its own structure and schema, even if semantics are the same. It would be a great advancement for each biologist and other interested people if all this data could be accessed from one single point and view. In return, each user or institute would certainly like to share its own knowledge in this way. If individual users can add personal knowledge, experiences, reports, and similar, a world-spanning repository can be deployed. Similar scenarios can be found in many other research areas, which are not only located in the field of natural sciences.

Scientific applications often make use of *Grid Computing*. Grid Computing combines cluster computing with the enormous power of the Internet community. The idea is to form a virtual super computer composed of networked machines. In contrast to cluster computing the participants are more loosely coupled, heterogeneous, and geographically dispersed. One the one hand, this is used to build computational grids, which utilize the processing power of the participants in order to process computationally intensive tasks. This has been applied successfully in scientific tasks, but also in commercial enterprises. A good example is the Search for Extraterrestrial Intelligence at home (SETI@home) project [13]. On the other hand, there exist grid computing systems that deal with sharing and management of large amounts of distributed data, so-called data grids. These systems are often combined with computational grids, because computationally intensive tasks often rely on large amounts of data. This is not at least another outcome of

the rapidly growing number of sensors used world-wide. An example is the Southern California Earthquake Center [186], which uses grid systems to research the phenomena of earth quakes in Southern California.

Data grid systems require widely distributed access to data by many people in many places. They experience great success in e-science applications [189]. Due to the controlled sharing and managing approach, data are rather homogeneous in these scenarios. But, data grids require sophisticated techniques for distributed query processing. Crucial challenges are the loosely-coupled character of these systems and the heterogeneity of the participants regarding processing power, storage space, and query processing capabilities.

A next step in this context – or just another view on it – are *cloud services*. Cloud services are usually hosted in data centers and provide software and computing facilities as a service. Cloud computing services were mainly developed due to economic factors and help to reduce the costs of ownership in the first line. Cloud computing is specifically challenging due to its sheer scale. Concerning query planning and processing, it is simply not feasible to process an exhaustive search space over thousands of sites. Cloud services are a rapidly emerging trend and there are strong efforts for providing large-scale storage services on the idea of cloud services, a sort of (simplified) cloud database services. We discuss this emergent trend in more detail in Section 3.1.

Semantic Reality Driven by the ideas of the Semantic Web, there is a clear trend to enrich application data (such as sensor readings, information about people and "things", knowledge articles, scientific publications, ...) by semantic metadata. This is a major prerequisite for enabling collaboration in a meaningful and practical fashion. The original idea, which is to describe Web services and sites by appropriate semantics, is adopted by many novel applications. For example, the idea of the Social Semantic Desktop [79] is to organize and share information from user desktops by utilizing metadata as proposed by Semantic Web standards. The project aims at connecting desktops from single users, preferably in an ad-hoc fashion. Due to the very high number of potential participants, and thus, huge amounts of desktop data and different desktop applications, an agreement on a global schema for metadata as well as any centralized repository is futile.

Another good example is the GSN framework [10], which proposes a large-scale extendible global sensor network composed by a multitude of smaller and separated sensor networks. Users may develop their own virtual sensors by accessing the output from existing (real or virtual) sensors. Like this, sensor data from all conceivable application domains and networks becomes accessible to all participating users. Connecting the sensor networks forms a sensored virtual world. Making the sensor readings (real or virtual) available to all interested users connects the virtual world with the real world – forming the *Semantic Reality* [64]. To make such an application work, adding semantics to sensor descriptions is a key enabling ingredient. This data describes characteristic attributes, like locality, type, etc. Only then the sensor descriptions can be indexed and searched meaningfully. Due to the high number of different sensor networks, sensor types, and designations, this data will be very large and heterogeneous in its structure. The mere amount of sensor data as well as semantic data will exceed the possibilities of any centralized infrastructure. This trend is already observable at popular Web search engines today. For example, with 6 billion web pages to index and millions of searches run daily, the size of storage managed by Google [75] probably was between two and five petabytes

in April 2003. Not at last the economical consequences from this drive the development of distributed Web search engines with increasing intensity [113] – another convenient application example. Concerning Semantic Reality, the accumulated sensor data from only cars of 2015 will probably exceed the 6 terabyte per second limit [168]. Here, the challenges discussed in this work primarily arise from implementing a system for dynamically indexing, managing, and querying the semantic metadata. Though, this does not reduce its capabilities of managing also the actual sensor data.

Knowledge Sharing The popularity of large-scale distributed data management among private Internet users originated in the development of file-sharing applications. Meanwhile, people can find almost everything related to computer systems, from digital music over movies to full-functional software packages. Trust, privacy, and not at last the infringement of copyright laws have been crucial problems from the beginning. Such systems usually apply the P2P paradigm up to different extents. Napster [144] supports data exchange directly between participants, but uses central servers for indexing data. In contrast, Gnutella [74] and Freenet [70] are absolutely distributed, from the indexing of data over search to the final exchange. Due to scalability issues [176], the query processing techniques used in early representatives had to be revised. Modern systems tend to replace the standard flooding approach (see Section 3.2) by directed routing decisions, which are based on distributed indexes and are backed by leveraging the structure in the network layer (see, e.g., [9]).

The enormous success of these applications inspires a lot of novel knowledge sharing systems. These systems usually claim for using semantic metadata in addition to raw user data. Heterogeneity (in both, data structure and processing capabilities), scalability, and robustness are primary issues. Examples can be found in the e-health domain. Popular approaches are doctor-to-doctor networks easing the exchange of patient data and remote patient monitoring. Diagnoses can be supported by searching for similar symptoms in other patient records, may be backed by new findings from biological, chemical, or psychological research, and finally made available to the community. Another example is to combine data from observatories all over the world [212]. The combined data may help other astronomers, but also astronauts, meteorologists, or people with private interests. This scenario shows a very special requirement, which is in parallel a particular interesting one: astronomers show special interest in ad-hoc collaboration and sharing of knowledge, e.g., if the Halley's comet is currently passing earth.

By extending this vision of collaboration we take the next step towards *Networked Knowledge* [64]. Why should such systems not combine knowledge across all research and application domains? A biologist active in research may often be interested in accessing data from chemical or mathematical data collections, but also in reports on current changes in law or politics. Astronomers will often be interested in current meteorological data, but maybe also on geological. If a doctor needs to reach a patient in an emergency situation, current traffic information will help him to decide about the best way for doing so. It is easy to extend this idea to the financial domain, telecommunications, disaster management, rescue missions, etc. The point is not to identify possible correlations and interactions and to build accordingly specialized solutions – the point is to enable the sharing of data and knowledge in a flexible, ad-hoc, and dynamic manner. As data

will be structured differently in different systems and across different domains, handling heterogeneity will become an even greater challenge.

There are a lot of other conceivable scenarios with similar requirements. For the remainder of this work, we summarize them all under the umbrella of Public Data Management. The proposed techniques and methods are suited for a wide variety of corresponding applications. Next, we discuss the actual challenges that arise in these scenarios.

Chapter 2

Problem Statement

The introduced application scenarios and their specific characteristics reveal challenges for a couple of research directions in IT. This spans from the physical level mainly concerning networks over general system engineering and design to information integration on data level. This work focuses on the challenges arising from information management, information sharing, and information filtering. This may be raw user data, or index data, or both. Section 2.1 discusses the main challenges and requirements for building application-independent large-scale distributed storage systems needed for completing this task. In Section 2.2 we lead over to the actual contributions of this work, particularly focusing on the issues concerning query processing.

2.1 Challenges

Some of the following challenges may be similar or refer to similar problems and key aspects. However, there exist different aspects between them or they have to be considered from a different point of view. We can roughly classify all the challenges along three questions:

(1) How to structure and organize data in massively distributed settings?

(2) What is necessary to build a robust and practical solution?

(3) How to query data and how to query efficiently?

The first question mainly deals with *data organization* and raises two challenges:

Genericity and Flexibility We consider structured data as a basic requirement for building information systems as introduced before. Due to the very high number of participants and data sources, trying to agree on a common global schema is not practical at all. Thus, we need an exceptionally generic and flexible schema for structuring and managing data. A main requirement in this context is extensibility: it should be easy to integrate new elements and new content without conflicts or the need for restructuring. In the end, we need a uniform and universal model, which is open and can adapt to changing content and structure. Moreover, this should also reflect in the way queries are

formulated. Users should not need to worry about whether they know specific schema elements or not.

Dealing with Heterogeneity The goal is to combine data from many different sources, in its largest extent these sources also come from many different domains. As the agreement on a global schema is unrealistic, there is a strong need for techniques resolving heterogeneities on schema level (different naming and structuring of the same concepts) and on data level (different representations for the same real-world objects). The intended systems are very large and loosely coupled. Thus, the creation of integrated views should happen ad-hoc and user-driven, allowing for a pay-as-you-go integration [133]. For this purpose, we need to support semantics for information-driven integration and appropriate modeling concepts like schema correspondences. A major requirement of a universal storage system is to handle these semantics as standard data. Heterogeneity also applies to the technical level. With such a large number of participants united, we have to expect different query processing capabilities. Thus, strategies and techniques for processing queries have to be adaptive and flexible as well.

Question (2) touches the *architecture* of a large-scale distributed platform and *practicability* issues. Among others, the main challenges are:

Scalability From our point of view, a centralized architecture could never accomplish all the tasks arising from the challenges listed here. Centralized management raises several problems, such as a single point of failure, limited scalability in terms of participating users and data, reduced resilience against attacks, and so on. Moreover, there exist economical arguments for decentralization: the load and costs of managing data and maintaining services. Providing data access to such huge numbers of participants can quickly exceed the capabilities of any single centralized provider. This is especially true if regarding systems that are even beyond the current size of the Internet. Thus, decentralized approaches are the only meaningful alternatives. The architecture should support self-organization and automatization, load-balancing, and preferably location transparency (users only search for data, without any idea of the actual location of this data). Scalability needs to be achieved on the levels of the number of participants and the amount of managed data. Scalability issues also arise from complex query operations, imports and updates, and maintenance operations in general.

Robustness and Availability A main requirement of distributed storage systems is to achieve resilience against node and link failures. One popular way to address this issue is to maintain redundant links and replicas of data objects. But, such approaches raise several other issues. This spans from the actual degree of replication over appropriate strategies for update propagation to problems of choosing among the set of replicas. In the end, replication is not the all-in-one solution for handling unreliable networks. Participating systems and links between them may fail during processing, some data or network parts may be completely unavailable at query time, messages may get lost, etc. This emphasizes the need for relying on the open world assumption, where no constant result sizes or similar can be guaranteed. The open world assumption implies querying in a best-effort manner. This means, results are “as good as possible”. Nevertheless,

distributed storage systems should be able to cope with node failures and other problems. Meaningful and significant results in all times and situations are a fundamental prerequisite.

Privacy, Trust, and Fairness Privacy of the plain data, their providers, as well as their users is a must for distributed storage systems. This involves the need for secure storage and transport, authentic and trustworthy sources, and resistance against attacks. For instance, a centralized architecture always has the smack of a "Big Brother" scenario. Another main requirement is the proper distribution of data and load. In first line, this fairness is for guaranteeing performance of the system, also in times of high load or churn (i.e., frequent membership changes). Second, the same rights of usage and accessibility must be provided to each user – in the opposite, each user must adhere to the same duties, responsibilities, and charges. A good counter-example are so-called free riders in file-sharing systems [12]. These users (by the way the corpus of most of the popular systems) only look for data, but are not willing to provide or share own data. This may result in hot-spots, performance leaks, and not at least in problems concerning trust and reputation.

The third question is related to *query formulation* and *query processing* with the following challenges:

Expressiveness of Queries The generic and flexible data model motivated above needs to be accompanied by an appropriate query language. For querying structured data, such a query language should support the core features of classical database query languages. Further, the query language has to support special query operators for dealing with large-scale data (e.g., ranking operators), heterogeneities at different levels (e.g., similarity operators), and unknown schemata (e.g., mapping operators). For this purpose, techniques and methods known from Information Retrieval (IR) [194] need to be adopted. Ranking queries allow to get an overview of large data sets, which can be refined iteratively. Similarity queries support the filtering and combination of data which is textual similar. This helps handling spelling errors and similar, but also to identify correspondences. Mapping operators allow for querying schema data, like attribute names and correspondences, and treat this as plain data. This list can be easily extended. Of course, this does not only concern the language level. The specific language constructs and concepts also need to be backed by according query operators. These operators require efficient and alternative implementations, which leads over to the last great challenge.

Efficient Querying As we argue above, meaningful scalability can only be achieved using a distributed and decentralized architecture. The same holds for the actual processing of queries. Using *data shipping* [120] the required data are located and shipped from the providing participants to the requesting one, where it is processed finally. In most cases the *query shipping* [120] approach will be much more appreciated: (parts of) queries are shipped to where the required data resides and are processed there. This requires distributed implementations of the supported query operators. These distributed operators should be combined with an appropriate adaptive query processing strategy

in order to be able to react to changing network situations, high load, failures, available knowledge, etc. Alternative implementations of the same logical operator should be available for usage in a cost-based optimization strategy. These are basic prerequisites to handle the dynamicity of the whole network and the autonomy of the individual nodes. Further, increased parallelism of query processing (up to a certain degree) is fundamental for processing complex queries with satisfying performance and meaningful results, particularly in the presence of peer failures and similar.

Finally, querying should be ad-hoc (like the integration of heterogeneities) and *in-situ*. This means a result is complete and correct if it contains all data from all participants available *in the current situation*. For instance, Google uses crawling processes to collect index data in regular intervals. These processes are very long-running, due to the high amount of data available in the Web. Despite this will get a problem concerning the steadily increasing amount of data, this is fine in principle due to only few highly dynamic data sources, e.g., news, weather, etc. Such services are known a-priori and thus, can be prioritized. In Public Data Management applications data are usually highly dynamic (also referred to as *The Deep Web*). Thus, the general problem of data freshness and accuracy becomes more important. An intuitive way is to provide and access the data instantaneously from where it is generated, i.e., ad-hoc from and by the participants of the system. From the distributed systems' point of view, this relates to the old pull vs. push / synchronous vs. asynchronous problem.

Guarantees Query processing should come along with exact guarantees for query time and overhead (in terms of messages and hops). Moreover, information on availability, actuality, completeness, and other advanced features should be provided. But, providing such guarantees in large-scale distributed systems is an extremely challenging task. Due to the very large scale and the participants' dynamic patterns of use, data and service quality can often be guaranteed only in a best-effort way. But, usually this is absolutely satisfying for the services and systems we aspire. This shows up in many services we already use on a daily basis, which follow this approach and still provide meaningful service (e.g., email, DNS). As we explain in Section 3.1, the amount of possible guarantees decreases with increasing autonomy of the participants. Nevertheless, database-like applications require guarantees. Thus, trade-offs between guarantees and assumed autonomy have to be identified and methods for providing probabilistic guarantees have to be established. The latter should be used in all situations where exact guarantees are not available. This particularly holds for information on the completeness of query results, which cannot be determined straightforward in the presence of adaptive and highly parallelized query processing strategies without any global coordination.

This work focuses on the one fundamental requirement that is common for all the introduced application scenarios. The following question also captures most of the deduced challenges: How to build a distributed universal storage system? This represents a variety of aspects far to large for being covered by a single work. Thus, we explicitly focus on the challenges of distributed query processing. Nevertheless, we need a foundation for this in terms of system design, architecture, data model, etc. Before we get over to the contributions, we summarize the main assumptions that hold throughout the work. We assume:

1. A totally distributed infrastructure, no central instance for coordination or similar.
2. Heterogeneously structured data, consisting of different parts provided by a large set of participants.
3. Database-like structured queries, which provide basic functionality for IR and other specialized tasks – but these tasks themselves are not in the focus.
4. Integrational tasks have to be supported. As the actual integration will depend on user preferences and application scenarios, a provision of tools and techniques for these tasks is more important than a discussion on the full depth of handling semantic correlations.
5. Data are completely physically distributed, in contrast to, for instance, Peer Data Management Systems (PDMS – see Section 4.3), where data are strictly located at the providing side.
6. There is no pronounced need for supporting concurrent transactions with strict update consistency.
7. We handle data of public interest, which is not enterprise data or similar. Thus, high-end real-time databases with exact guarantees concerning availability, query times, etc. are not mandatory. Among others, applying the open world assumption satisfies the users' needs.
8. We expect no privacy issues and the willingness of all participants to act in a fair and honest manner.

Note that assumption 5 does not prevent from building applications that regard data location as fixed. In these cases, the proposed techniques are tailor-made for providing an index structure where only the index items are distributed. Assumption 6 is for narrowing the focus of this work. Nevertheless, transaction and update handling is an important issue for general-purpose database applications.

2.2 Contribution and Structure

There has been much work on many fields concerned by the ambitious vision of building a universal distributed storage at Internet scale and above. On the network level, distributed architectures like Grid and P2P systems have been exhaustively investigated – and still are. On data level, we can inherit a wide variety of the contributions from traditional and federated database systems as well as the deep research in the field of IR. However, today most of the introduced challenges and problems are only solved to a small fractional amount. Among the main challenges, this work lays its focus specifically on those arising from information searching and filtering. This comprises the formulation of queries, but focuses on the technical aspects of actual query processing. In this section, we list the main contributions of the work in hand.

The list of contributions is:

1. We achieve a scalable and robust infrastructure providing data availability and actuality by relying on the P2P paradigm.
2. We follow a strictly data-centric approach and introduce a triple-based data model that is particularly flexible and generic.
3. Based on this foundation, we propose several query operators implementing query functionalities tailor-made for large scale heterogeneous data collections.
4. These operators are used in queries formulated from a user's local point of view and allow for leveraging expressiveness of queries.
5. We discuss several physical implementations of these operators and propose novel ways for utilizing the indexing facilities of the underlying P2P system for that.
6. Resulting query plans are integrated into a query engine that follows modern principles for efficient distributed query processing. We implemented a truly scalable and robust storage system on the basis of this query engine.
7. Further, we provide specialized, but tightly integrated, capabilities for handling semantic data representing schema correspondences.
8. Finally, we argue on guarantees and predictability, focusing on the completeness of query results.

In one sentence, we propose novel techniques for enabling generic Public Data Management applications with database-like features, whereby all occurring costs are shared by all participants. In recent times, P2P techniques have proven to be perfectly suited for building large-scale distributed systems, particularly because of their implicit support of collaboration. This is, among others, highlighted by the fact that P2P systems consumed about 74% of overall Internet bandwidth in 2007[1]. Inherently, P2P systems tend to follow a best-effort approach. This is only one of their many facets and advantages. By their nature, they also help to overcome problems of privacy, because there is no central instance and there exists no single user who may obtain global knowledge. In contrast, they raise new problems of trust and fairness, due to the unpredictability of user behavior and trustworthiness. In Part II we first introduce the state of the art concerning large-scale distributed systems, before we introduce the P2P paradigm in more detail. This is matter of Chapter 3, which additionally introduces the general architecture that we propose. Afterwards, in Chapter 4, we provide insights into the P2P-based storage layer and explain how to model data for supporting a universal and flexible storage.

The challenges of achieving stateless, parallel, and efficient query processing are matter of Part III. We start with discussing techniques for appropriate query formulation in Chapter 5. The introduced algebra provides a basis for explaining implementation variants of operators and according processing strategies in Chapter 6. Chapter 7 finally deals with the crucial challenges of providing guarantees for query processing, with a focus on estimating the completeness of query results.

[1] http://www.ipoque.com/news_&_events/internet_studies/internet_study_2007

The proposed concepts, strategies, and techniques are implemented in a prototype system. The implementation allows for a proof of the concepts and is a running and practicable foundation for future work. We implemented a light-weight infrastructure, providing an interface for managing and querying data in large-scale distributed universal storage systems. An overview of implementation issues is matter of Chapter 8. We use this implementation for proving the suitability and arguing on the performance of the proposed methodologies and techniques on an empirical basis. A corresponding evaluation is also presented in Chapter 8. After that main chapter of Part IV, we conclude the work in Chapter 9.

Part II

Large-Scale Distributed Data Management

Chapter 3

Architectural Approaches

This chapter presents a brief summary of historical developments and insights into the state of the art concerning large-scale distributed systems. Section 3.1 deals with some related work in this context by covering general approaches for building large-scale data systems. We take a closer look on current trends like cloud services and classify them accordingly. That section helps to further separate the work in hand from other system approaches. Afterwards, Section 3.2 introduces main facts about the P2P paradigm. We give a brief classification of P2P systems and motivate why they are the choice in this work. Finally, we present the proposed layered system architecture in Section 3.3.

3.1 State of the Art

Distributed systems have been all around for a long time now. With increasingly computationally expensive tasks, the idea of combining resources distributed over multiple participants became more and more popular. The standard textbook describing different approaches and necessary trade-offs is [200].

In Figure 3.1, we classify all the approaches we mention in the following along the three dimensions *distribution*, *autonomy*, and *heterogeneity*. This classification is inspired by [32], but we slightly extend their original scheme. We indicate where the different system approaches can be arranged. Note that this is only a coarse grading. Some of the systems may also be located in the nearby neighborhood of what the points indicate in the figure. For instance, there exist MDBSs (Multi-Database Systems) that apply a higher degree of distribution than shown. Further, PDMS based on super-peer architectures show a degree of distribution between client-server and pure P2P. The work in hand focuses on what we call *P2P databases*, which are strictly decentralized, pure P2P, and assume a high degree of autonomy, but can still provide guarantees. P2P databases based on DHTs can support the principle of logical data independence to its full extent. Traditionally, this principle claims that the logical model should be independent from its physical representation. In distributed systems, data should also be independent from their physical location.

Another crucial criterion for classifying distributed data systems is the provision of guarantees. This spans from guaranteed (maximal) costs over availability to completeness and soundness of results. Guarantees can be identified as a fourth dimension. But, as a main observation from all the works dealing with large-scale distributed systems, the

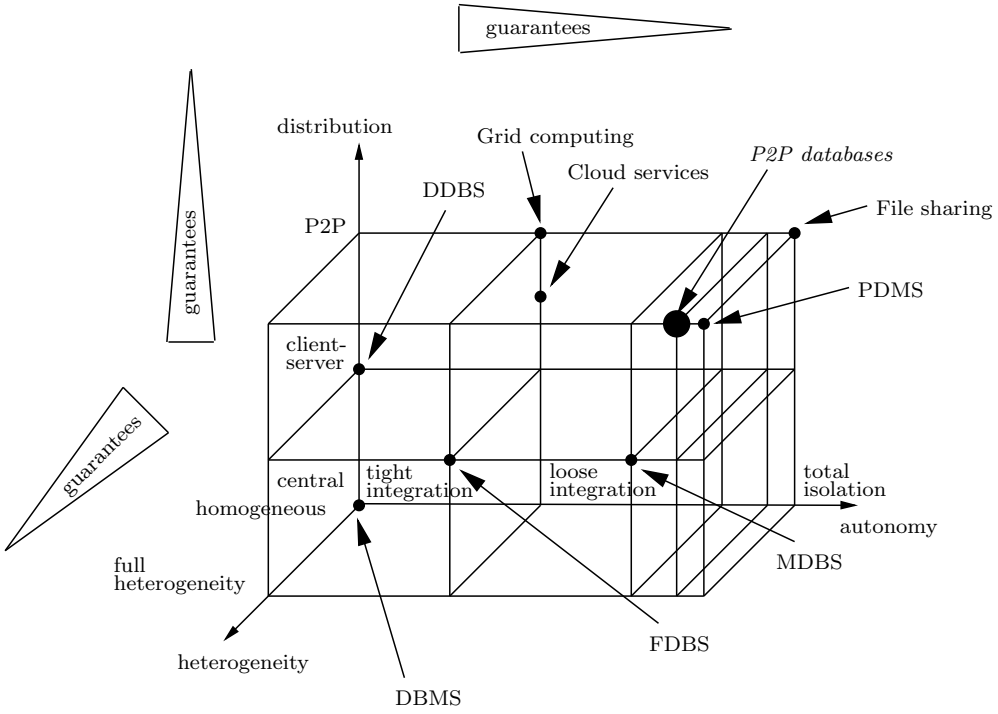


Figure 3.1: Classification of distributed systems

degree of possible guarantees decreases with each of the other three dimensions increasing. For instance, the more autonomous the participating nodes are, the less guarantees can be provided. We indicate this in the figure as well. PDMS can give exact guarantees only concerning query overhead due to a used time-to-live (TTL) limit. All other guarantees can be only probabilistic, which is in contrast to P2P databases that can also provide guarantees in terms of completeness and lookup costs. This is due to the increased autonomy of peers in PDMS compared to peers in P2P databases.

Database Systems Distributed computing started with combining several computers to form a cluster of machines. With this, people were able to process computationally expensive tasks with significantly increased performance. The idea of distribution was applied not only to processing power. From the database perspective, the first steps towards distribution were taken by partitioned databases. Here, data (and optionally control) is distributed over several participating machines. Usually, partitioning features come as an extension or feature of traditional databases, such as Oracle [158] or IBM's DB2 [61]. Relations are split into smaller fragments and distributed over different sites, which is called data fragmentation or partitioning. There are two general approaches for that, horizontal partitioning [46] (split into fragments of tuples/rows) and vertical partitioning [149] (split into fragments of attributes/columns). Also hybrid approaches combining both schemes developed over the time. This evolved to the idea of distributed databases [159]. A distributed database system (DDBS) consists of several logically related databases in a network. DDBS also apply data partitioning. In a DDBS the partitioning scheme is globally known, either at a central control instance or at all par-

ticipating sites. A federated database system (FDBS) [191] is a collection of pre-existing database systems. Operations are applied on multiple component systems in a coordinated manner. Usually, the components are heterogeneous concerning structure and data schemata. For integration, FDBS rely on a single federated schema.

With the rise of the Internet new efforts were made to build "even more distributed" systems at large scale. First initiatives headed towards mediator systems [184] or multidatabase systems (MDBS) [34]. MDBS are similar to FDBS, but multiple federated data schemata can exist. MDBS allow an increased autonomy of the components. Both, FDBS and MDBS, may contain DDBS as components. All these systems rely on a common federated schema that integrates the local components, either defined a priori or as a result of global agreement. lack in scalability and flexibility due to centralized facets. In FDBS and MDBS, mappings express the correspondences between local schemata and global schemata, where the Local-As-View or the Global-As-View [83] approach is applied. With proceeding time, the popular client-server model was stressed to the max. In this model one or several servers process the tasks requested by a multitude of clients. For achieving true scalability, systems are forced to distribute load, responsibility, and charge more and more. The most promising approaches are those aiming for decentralization. The strict implementation of decentralization originated in the idea of the P2P paradigm, which is introduced in the next section.

Cloud Services Concerning large-scale data collections the Internet is a perfect example again. The wealth of information that is, for instance, handled by Web search engines like Google [75] is incredible. Thus, the content of the Internet itself represents the potentially largest data collection available these days. Following, there are several developments that propose new ways for handling such huge amounts of data and participants. Recently, Google researchers developed a technique called *MapReduce* [62], where clustered machines are used to process Web data efficiently. The basic concept builds on the map (map intermediate solutions to a key) and reduce (reduce lists of intermediate results to final results) functions known from functional programming, although they slightly differ in their semantics. The benefit is that all map and reduce computations can be processed concurrently. Thus, it can be understood as a kind of low-level parallel programming paradigm – without demanding special parallel programming skills from developers. Actually, MapReduce reinvents an idea more than 20 years old and neglects some valuable experience unfolded by database research over the years [1]. Nevertheless, it shows significant success and popularity these days due to its excellent scalability. The approach can be mapped to a lot of computational problems. From a conceptual view, the techniques for processing query plans proposed in this work can be seen as an implementation of the MapReduce principle. But, while MapReduce focuses on efficiently processing computationally expensive tasks on extremely large datasets, we aim at utilizing distribution of data and load by adhering to a strictly decentralized processing without any central coordination. In conjunction with all the turbulences and discussions around MapReduce and cloud computing, there are several other recent developments aiming at large-scale data systems. Based on the Google file system [72], Google BigTable [48] is an extremely large-scale proprietary database system used in many Google applications.

[1] http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html

BigTable departs from some conventions of traditional database systems and implements a mixture of row-oriented and column-oriented approaches. It is designed to scale from hundreds to thousands of cluster machines. Thus, it is rather a large-scale DDBS than a P2P storage system, as proposed in this work. There are some open source efforts to implement Google's ideas around MapReduce and BigTable, e.g., in Apache's Hadoop project [82]. One subproject of Hadoop is HBase [86], a large-scale database modeled after Google's BigTable. Amazon Dynamo [63] combines ideas of distributed hash tables (DHTs, see Section 3.2.2) with other techniques known from operating systems and distributed systems research. Dynamo was developed to provide an incrementally scalable and highly available key-value storage. This storage solution is used by another large-scale distributed database, Amazon's SimpleDB [193], which is in turn used by a couple of other web services. It works close with the online storage service Amazon S3 [181]. SimpleDB supports simple queries on structured data and provides a view on the data similar to that known from relational databases. Amazon S3, in turn, provides a rather simple and pure storage service for web developers, which enables them to use the same powerful storage solutions as Amazon does for their own services. There exist also academic efforts in this direction. OceanStore [124] is such a project running at the University of Berkeley. OceanStore provides an architecture for global-scale persistent storage. As the works mentioned before, OceanStore follows a utility model. The authors focus on general large-scale data management as well as security, privacy, and update processing. They do not discuss database-like query processing in detail. Further, OceanStore is designed as a storage service for single data sources, whereas we discuss methods to integrate heterogeneous data sources into a universal storage at Internet scale. Finally, we have to mention PNUTS [164] in this line of services. PNUTS is a platform for distributed universal storage developed by another big player in Web business, Yahoo!. PNUTS aims at a simplified database-like storage service for extremely large datasets. It uses a DHT on its physical layer. All these systems are primarily designed to support Web search and other tasks related to the field of IR [194]. IR tasks imply a different conceptual view and target different challenges than the work presented here. This is why the systems follow a rather pragmatic approach having its seeds in the field of distributed systems research – in contrast to the database view we propose. So, all of them come with simplified querying mechanisms, which often only provide a simple put/get interface. The work in hand, in contrast, discusses approaches for supporting full-fledged database queries, enriched by operators tailor-made for IR tasks (e.g., ranking queries). Thus, like all these distributed systems aim at providing cloud computing and storage services, we aim at providing a kind of cloud database service. Such a service enables the participants to manage, publish, and subscribe to large-scale datasets in the same way as they are used to do it with traditional databases. Moreover, most of the aforementioned systems are built to run in a cluster of machines, where still some central coordination and global knowledge is available, needed, and expected. The work in hand rather focuses on a large-scale universal storage that is totally decentralized and in which participants act absolutely autonomously. Thus, the methods and techniques proposed here are also suited for leveraging the functionality (and even scalability) of existing cloud services to a database-like level.

Peer Data Management Systems Several proposals deal with the notion of PDMS [84, 15, 92, 179]. There are different definitions for PDMS, but all are based on the same principles. We follow the notion presented in [95]. The authors classify PDMS as large-scale data systems with autonomous schema-based data sources (peers). In contrast to our work, the integrational aspect of different data schemata and schema mappings is the main focus of PDMS. Rewriting of queries conforming to the established schema mappings is a primary issue. In PDMS data resides at those peers that also provide it, because they are usually using an unstructured P2P overlay. There are also approaches using hybrid P2P overlay networks [39]. Data integration is achieved by pair-wise interactions without any central coordination and global schema. Query processing is usually based on a kind of (advanced) flooding. In order to increase scalability and efficiency, some systems use specialized routing indexes [56]. Routing indexes are usually established on instance and schema level. Our approach is to handle schema heterogeneities by inserting semantic metadata in the same format as raw user data. We suggest to provide automatic query rewriting as a feature rather than as a prerequisite (see Sections 4.2.3 and 5.1.2). PDMS are also called schema-based P2P systems. To distinguish our approach from PDMS, we use the term P2P databases. From this point of view, PDMS are data integration systems focusing on mappings and query rewriting, whereas P2P databases refer to distributed indexes on the basis of scalable distributed data structures (SDDS) [150]. [32] presents database-centric and P2P-centric features that underline this differentiation.

ACID vs. BASE and the CAP Theorem The term P2P databases implies several crucial features of traditional database management systems (DBMS), such as the ACID principle, consistent updates, security etc. As this work focuses on query processing, we do not discuss these issues in detail. Nevertheless, a brief statement on this is advisable. There is the unanimous insight that large-scale data systems will always have to struggle with the trade-off between availability and consistency. This is captured by the CAP theorem [73]. This theorem states that any shared-data system can have only two of the three CAP properties:

C onsistency

A vailabilty

P artition tolerance

[73] proves this theorem and gives suggestions how to handle this issue in practice. In our approach, partition tolerance is implicit, as this comes as a feature of the DHT used on the physical layer. All large-scale distributed data systems should aim at availability as the second property. This is expressed in the BASE principle, which mostly displaces the ACID principle from traditional databases. BASE stands for:

B asically A vailable

S oft-state

E ventual consistent

This means, we have to sacrifice consistency and isolation (the C and I in ACID) in order to increase scalability, performance, and robustness. This is also what Amazon, Google, etc. do in their storage services. In contrast to them, by supporting database-like structured queries we also detach from rigid simplicity, which is a key aspect of, for instance, the MapReduce approach. Query results always correspond to a situation of a specific point in time, which may be "historical". This is why this best-effort approach is also called in-situ query processing. The CAP theorem also helps to distinguish P2P databases from central and cluster databases (forfeiting partition tolerance) and distributed databases (forfeiting availability) [38]. Note that most large-scale distributed data systems choose to relax consistency rather than availability and partition tolerance. Most Web applications do not need strict ACID properties [207], whereas availability and robustness are a must. We expect this as a general rule for Public Data Management. Nevertheless, consistent updates are a main requirement for implementing a general-purpose database application. If they are required, they have to be implemented on top of the underlying data system as part of the application. Most of the works listed here assume read operations as the majority of actions. But, guaranteeing consistency at any time on the basis of advanced replication and transaction protocols gains more and more attention these days. First works considering this for P2P databases are, for instance, [35, 114, 63]. Following from the CAP theorem, systems supporting increased consistency will inevitably end in decreased availability, performance, or robustness. In our work, we do not abandon consistency. Rather, due to the CAP theorem, we have to relax it and aim at eventual consistency as proposed in [63].

We believe in the strengths of the P2P paradigm to realize widely distributed data systems on really large scale in both, managed data and participating nodes. All P2P approaches usually have to sacrifice certain ACID properties (see the CAP theorem) and apply the open world assumption. Systems based on a lower degree of distribution usually apply the closed world assumption. In the above classification, the term P2P is used to represent the degree of full-scale distribution. However, there are different P2P-based architectures that could be differentiated in the degree of distribution on an even finer granularity. Next, we give more insights in the characteristics of P2P systems and the differences between existing general approaches. [32] introduces further criteria for classifying P2P approaches in more detail. These criteria are classified into DB-centric and P2P-centric features. In the following, we highlight the differences from the perspective of these dimensions. For more explanations on this we refer to [32].

3.2 The P2P Paradigm

With the introduced developments in IT, the traditional client-server model revealed several weaknesses. Significant problems showed up in achieving meaningful scalability and robustness. The P2P approach evolved as an alternative to this. The main concept is to share resources in a cooperative way. Through this cooperation, systems with huge capacities (storage, CPU, bandwidth, ...) can be build – larger than any capacity that one of the peers alone could provide. In the idea of P2P, each participating node takes the

roles of a client *and* a server in parallel. This means, each node may act like a client requesting for service, data, etc. – and in return acts like a server processing analog requests from other nodes. Consequently, there are no clients nor servers anymore. Instead, all participating nodes are called *peers*. In early systems the nodes were called servents [74], a combination of the words serv(er) and (cli)ent. All services and all data can be used by all peers. Further, the P2P paradigm neglects any central instance, control, and knowledge. All functionality is provided by a strict pair-wise interaction between peers. The systems are characterized by strict self organization. Usually, a peer only knows about itself and its neighbors. Neighbors are defined by direct connections established between peers. This does not directly relate to the physical connections between peers. Rather, P2P systems are usually build as an overlay network on top of physical networks and protocols (e.g., TCP/IP). Thus, connections between neighbors are defined on a logical level, i.e., between two neighbors there may be several physical nodes that are responsible only for routing messages. In a P2P system, peers decide and behave autonomously. These characteristics, in conjunction with appropriate techniques for compensating unreliable behavior (e.g., replication protocols), result in excellent scalability and strong robustness against failures and attacks. [8] summarizes the main principles of P2P systems as follows:

- all peers are equal and can access the resources of all other peers
- communication is symmetric
- peers are autonomous
- peers and connections are unreliable
- each peer "pays" for its participation by providing access to its resources to other peers
- there is neither a central coordination nor a central repository
- no peer has a global view on the system, i.e., there exists no global knowledge
- the global behavior of the system is a result of local interaction between peers

If P2P overlays are used for building distributed data systems, certain traditional database principles will usually be relaxed. This includes transaction properties and central query planning, which usually claim for central coordination. Recent efforts aim at implementing more and more database-like features for P2P systems without breaking the P2P paradigm. Corresponding contributions of the work in hand are techniques for efficiently processing structured queries using strictly decentralized approaches. Similar to distributed DBMS, P2P systems rely on the concepts of data sharing and data partitioning. In contrast to them, peers in a P2P system are absolutely autonomous. There is no central coordination.

The general idea of P2P has been applied successfully in several systems, even if most of them adopt it only on single layers. For instance, Ebay [66] applies P2P only on the user level: users interact pair-wise to exchange goods and money, but all other actions take place in a centralized instance. In the file sharing system Napster [144] files

are exchanged between users in a P2P manner. But, search and location are still based on a centralized index facility. Maybe Gnutella [74] and Freenet [70] can be regarded as the first "real" P2P systems, because their architectures adopt the P2P approach throughout all layers. In recent times, P2P conquered a wide variety of applications, far beyond the scope of only sharing multimedia data like music files. For instance, Zattoo [219] provides video streaming by combining resources of all participants in a strict P2P manner. P2P systems also have success in other domains mentioned before, such as sensor data management, stream processing, and grid systems [125].

Besides the many new challenges that the P2P paradigm arises, there are as many opportunities and advantages:

- due to the abandonment of any central instance, the systems scale well in terms of participants and data mounts
- data are accessed directly at the source, i.e., it is always fresh
- systems become robust due to their decentralized architecture, there are no single points of failures or hot-spots suited for concentrated attacks
- not only due to the autonomy of peers, P2P systems are tailor-made for building reliable ad-hoc networks even in very dynamic environments
- to join an existing network, new peers usually have to know about only one peer already participating (the so-called bootstrap peer)

Following, by relying on the P2P paradigm we can build storage systems that exploit the gigantic storage and processing capacity of the available Internet nodes in the same way as the network layer exploits the worldwide communication devices for routing messages between nodes. For further details we refer to relevant literature like [8]. In the next sections, we give a brief classification of P2P systems. We focus on the degree of structure in these systems, because this is particularly relevant for this work.

3.2.1 Unstructured P2P Systems

In their first incarnations P2P systems formed loosely coupled networks and did not rely on any specific structure or topology [74,53]. Thus, these systems are called *unstructured*. Peers are organized in a random graph. The actual structure (and thus, characteristics like the diameter of the graph and degree of peers) results from a rather "natural" way of connecting peers. This may be based on purely randomized approaches, but may also include aspects like semantical distance or similar clustering models. Data and index items remain at the providing peers – there is no reorganization, no global structure, no central coordination. User queries are usually resolved by simple message broadcasts, which is called *flooding* the network. In this approach each peer forwards a received query to all of its neighbors, which results in a message explosion with ongoing search. Usually, queries are processed using a maximal time-to-live (TTL) parameter, which limits the search effort. At each involved peer the query is evaluated against a local index to find relevant data. Finally, if the TTL is not exceeded, the query will be forwarded. Unfortunately, results are often incomplete, because not all relevant peers could be identified in the valid

TTL. Nevertheless, flooding is suitable for a lot of applications, e.g., file sharing. This is due to the fact that files are often highly replicated. Usually, they are the more replicated the more popular they are. Thus, the naive flooding approach is suited very well to find them. Following, rare content is rather difficult to find using this query processing approach. To reduce the overhead and to increase completeness of query results, so-called routing indexes [56] were proposed, which allow to query in a data-oriented way. Techniques utilizing semantic relationships and trying to adopt small-world properties [115] can help to increase completeness and relevance of results even more. Still, unstructured P2P systems are not suited for providing guarantees. Moreover, the systems are limited in their scalability concerning bandwidth consumption, network maintenance, and query processing capabilities [176]. For more details on unstructured P2P systems we refer the interested reader to [132].

3.2.2 Structured P2P Systems

Confronted with limited scalability and issues about query completeness, P2P systems have been enriched by some kind of structure. Peer identifiers and resource identifiers are mapped to an application-specific identifier space, which is usually achieved using a hash function. Resources are assigned to peers on the basis of some distance metric. Structured P2P systems represent a consistent implementation of scalable distributed data structures (SDDS) [150]. A decentralized index is built by partitioning data horizontally. Peers are organized in a graph corresponding to the structure of the SDDS. This results in a graph embedded into the identifier space – the structure of the system. This structure is built through pair-wise interaction without any central coordination. It is used to route queries more efficiently and to achieve complete results, but comes on the costs of increased maintenance. Algorithms for distributed maintenance have to ensure the consistency of the distributed index.

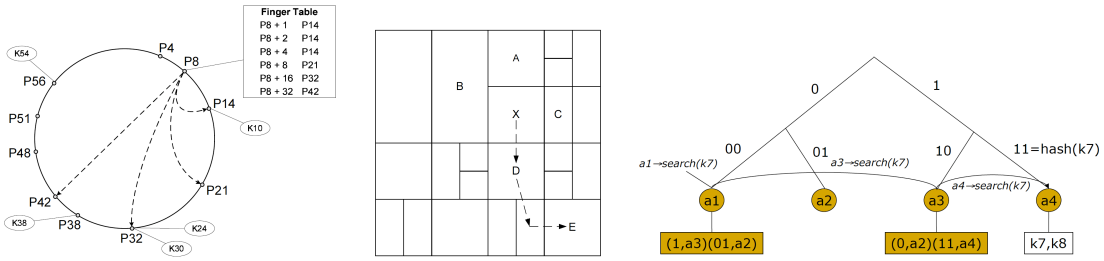
DHT systems offer only two operations:

- $put(k, v)$: insert the data item v using hash key k
- $get(k)$: return the data item(s) referred by the hash key k

Usually, the terms structured P2P systems and *Distributed Hash Tables* (DHTs) are used as synonyms. In [188] the author presents a more detailed classification. In the following, we list some of the most popular structured P2P systems and align them along this classification.

On the one hand, there are structured P2P systems using uniform hashing [196, 180, 220, 169]. Applying a uniform random hash function results in a very small probability that two different data items are mapped to the same hash key. These are the "real" DHTs, because they follow the principles of hash indexes known from traditional databases. Similar, they are designed for efficient single key lookups, but show rather bad performance for queries relying on data correlations, such as range queries. A uniform hashing produces random keys, which results in the index items being uniformly distributed. This implies a (theoretical) load balancing between the peers, but related data items are in general mapped to non-similar hash keys.

On the other hand, [188] distinguishes structured P2P overlays using order-preserving hashing [29, 85, 1]. In contrast to systems from the first category, they preserve semantic



(a) CHORD ring [188] (b) CAN grid [188] (c) P-Grid tree

Figure 3.2: Example topologies in structured P2P overlays

relations of resources when mapping them to the identifier space. An example is the prefix-preserving hashing used in P-Grid [1]: if resource r_1 is a prefix of resource r_2, then the hash key $h(r_1)$ will be a prefix of the hash key $h(r_2)$. This enables the efficient implementation of, for instance, range queries. DHTs from the first category have been extended by order-preserving hash functions as well, e.g., [80] extends CAN [169] in this way. Structured P2P overlays on order-preserving hashing can be compared to tree-structured indexes like B-trees. Due to the non-uniform hashing, maintenance efforts needed for achieving load balancing are increased. Many data items may be mapped to relatively small key ranges, because the keys follow the same distribution as the resources.

Besides the used hash function, the main difference between structured P2P overlays is the actual internal structure. For instance, CHORD [196] arranges the peers in a ring, CAN [169] uses a d-dimensional grid, and P-Grid [1] is based on a binary prefix tree. Note that the structure usually is only virtual. The P-Grid tree is not materialized as a tree on any peer – rather, it results from the key partitions assigned to each peer. Figure 3.2 illustrates the three mentioned structures in conjunction with established short and long range links. These links are used to route queries efficiently. The routing method of each system is indicated in the figure, too.

The exact routing technique is the next difference between structured P2P overlays, although it is based on principles common for all. All systems implement a data-centered greedy multi-hop routing. First, a hash key of a requested data item is determined. Then, the query message is routed towards the responsible peer by forwarding it to the peer as close as possible to the determined hash key. If available, peers choose between short range and long range links to fulfill this task. This follows the idea of small-world networks [115] and usually results in logarithmic message and hop complexity. During the routing process, peers choose among neighbors by looking up a local routing table (or finger table, as it is called in CHORD).

Confirming to [89], the design principles of structured P2P overlay networks are:

1. Small degree of each peer, i.e., a small number of direct neighbors.
2. Low latency, i.e., small diameter of the network.
3. Greedy routing without global knowledge.
4. Robustness.

Replication is a popular method to increase fault tolerance and reliability. Structured P2P systems provide several choices of where to apply replication: building several distributed indexes (which results in several overlays), using different hash keys for the same data item, or using multiple peers responsible for the same partition of the identifier space.

In this work, we propose to use a structured P2P overlay on storage level. The main reason for this choice is that these systems are able to provide guarantees – concerning completeness of queries (using sophisticated replication and maintenance strategies) and in the number of messages and hops needed for routing queries (usually logarithmic in the number of peers). This allows for cost-based query planning and efficient query processing. Nevertheless, we have to follow a best-effort approach in query processing, because large fractions of all peers and whole subnetworks may fail at query time. Structured overlays can guarantee robustness only up to a certain degree of failures. Another fundamental characteristic of all structured P2P overlay networks is that data does not reside at those participants that insert it. Rather, it is scattered over all peers that form the distributed index. If privacy of data is an issue, only insensitive index data should be managed in the storage system. Besides that, using the index as a full storage that contains also the raw user data is unproblematic.

The proposed techniques can be implemented on top of any of the introduced systems, because basically we rely on only the put/get facilities. Nevertheless, systems using order-preserving hashing (and thus, supporting efficient range query processing) show some significant advantages. For convenience, throughout this work we use the term DHT for a structured P2P system from any of the two introduced classes. Lookup mechanisms of structured P2P systems are surveyed in [19]. This work introduces principles of the most popular DHTs. For more details on structured P2P systems and popular representatives we refer to [188, 89]. [188] also highlights the benefits of structured overlays using order-preserving hashing for building P2P databases on top of them. We explicitly list the advantages of DHTs for our approach in Section 4.1. In that section we describe the physical layer of our approach in more detail.

Concerning the classification of distributed systems, DHTs are located on the same level of distribution as unstructured P2P systems are. A difference exists in the degree of autonomy that is assumed. So, PDMS, which are based on unstructured P2P systems, allow for a higher degree of autonomy than P2P databases do, which usually apply DHT technology. From the perspective of fragmentation and allocation of data, unstructured P2P systems follow a bottom-up integrational approach similar to FDBS and MDBS. In contrast, DHTs apply a dynamic system-defined fragmentation scheme. This is similar to DDBS, though in these systems fragmentation is static and user-defined. DHTs extend the query expressiveness provided in traditional unstructured P2P systems by advanced features like range and prefix queries. Nevertheless, both kinds of P2P systems still lack in a wide range of database-like querying features.

3.2.3 Super-Peer Architectures

Finally, there exist hybrid systems combining aspects from both, unstructured and structured P2P systems. This mainly evolved from the scalability problems observed in unstructured P2P systems [119]. Popular unstructured networks have been enriched by

hybrid features (e.g., Gnutella [74]) and other systems adopted this approach from the beginning (e.g., KaZaA [111]).

A popular example are super-peer architectures. The idea is to have a couple of dedicated peers, the super-peers, responsible for data-centric routing of messages. Usually, super-peers are nodes that are particularly powerful, reliable, or both. The bulk of standard peers is usually characterized by frequent joins, leaves, and rather short online times. Due to their increased reliability, super-peers are suited to establish a topology similar to structured P2P systems. This structure is used to route queries to those super-peers responsible for the relevant data – or with responsible peers in their neighborhood. Standard peers connect to a chosen super-peer and hence join their neighborhood. The super-peer usually has some information about the data available in its neighborhood, e.g., by building appropriate indexes. When a query is received by a responsible super-peer, the query is forwarded to the relevant peers in its neighborhood using flooding. An example is the Edutella [151] system. Edutella is a super-peer PDMS. It arranges its super-peers in a d-dimensional torus and establishes special index structures between super-peers and between super-peers and standard peers.

The techniques and methods deployed in this work can be applied to super-peer systems straightforward. They can be utilized for leveraging performance and efficiency on the super-peer level. All action taking place in the unstructured neighborhood of a super-peer is based on simple flooding, optionally with the usage of indexes. As this accounts for only a small fraction of all processing costs, hybrid systems can greatly benefit from the sophisticated optimization and processing techniques.

3.3 A Layered Architecture

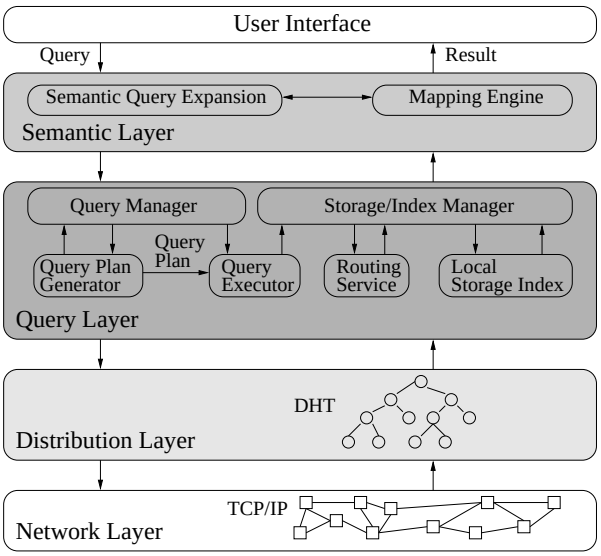


Figure 3.3: Layered architecture of a large-scale universal storage

In this work, we apply aspects from all of the mentioned approaches. From the explanations so far, we deduce that a layered architecture is best suited to utilize the power

of existing distributed approaches. We propose an architecture based on four layers as shown in Figure 3.3. The shaded layers indicate the focus of this work. The darker the layer is, the more we focus on it.

On the bottom *network layer*, we rely on the strengths of the Internet infrastructure by using only TCP/IP functionality. On this basis, the *distribution layer* provides support for transparently distributing data, query load, and index structures over the network by adhering to the P2P paradigm. This allows to exploit scalability, location transparency, logarithmic search complexity, and guarantees offered by DHT infrastructures [4]. The *query layer* offers storage, indexing, and querying of data on local and network data. We support a flexible and universal storage. Several indexes are built using the capabilities of the distribution layer. Query processing is provided by a set of database-like operators working on these indexes. All operators completely rely on the DHT features of the distribution layer. The *semantic layer* enables virtual grouping of data as well as creating mappings. Mappings are necessary for establishing semantic connections between different user schemata in order to share data and to integrate data sources. Creation of such mappings is easier if schemata are defined clearly. To support this, technologies from the Semantic Web are exploited in the semantic layer to represent different concepts and the relations between them. The user interface on top of all layers provides transparent access for the user from a local point of view. A structured declarative query language particularly supporting the handling of large and possibly heterogeneous data sets has to be supported.

Chapter 4

Distribution Layer

This chapter introduces the distribution layer of the proposed architecture. In Section 4.1, we summarize the advantages of DHT systems on that layer. This highlights how to utilize scalability, location transparency, logarithmic search complexity, and further guarantees like robustness and availability as they are provided by a DHT. We define an interface to the query layer by listing assumptions and requirements on DHT systems. P-Grid, the DHT that is used as a basis for all subsequent explanations and the implemented prototype, is overviewed in Section 4.1.1. To comprehensively solve the challenges from above, this basic layer has to be accompanied by according query processing approaches. This is matter of the subsequent chapters.

The crucial challenge is how to model data in a generic and flexible way. This concerns the applied data model and the distribution scheme. From a conceptual view, we combine aspects from the RDF triple model with the universal relation model [135] developed in the 1980s. The Resource Description Framework [171] (RDF) and the accompanying RDF Schema language [172] are agreed-upon standards for modeling and representing data in the Semantic Web. RDF was designed to describe Web resources, but is meanwhile used as a general approach for modeling information like metadata. We adopt the RDF approach because of its strengths in supporting (loosely) structured and heterogeneous data. This combination allows us to create a model for managing and representing data confirming to the requirements of universality, flexibility, and extensibility. The triple model is introduced Section 4.2. We list advantages and disadvantages and present the fundamentals needed to focus on query processing later on.

Structured data and according data models are used in almost all data-centric applications, not only in the Web. Maybe the best investigated is the relational model known from database systems, with its clearly defined structures, semantics, and integrity constraints. With the development of modern Web applications semi-structured data like XML [216] began its triumphal procession. XML supports flexibility and extensibility in an exceptional way. It became the de facto standard for exchanging data in Web applications and is the preferred choice in many distributed storage systems. Thus, we focus on how to apply the introduced triple model to relational and XML data in Sections 4.2.1 and 4.2.2, respectively. Schema correspondences are mandatory for dealing with heterogeneity. Last but not least, Section 4.2.3 discusses how to model this metadata in the way that is also used to model raw user data. This provides the basis for handling

heterogeneities in the semantic layer by using the functionality of the query layer. The chapter is concluded by a discussion of related work in Section 4.3.

4.1 DHTs in the Distribution Layer

As outlined before, we propose a layered architecture. At the very bottom, we rely on the TCP/IP protocol. This basically exploits the gigantic capacities of today's Web. On top of this we use a DHT overlay. By this, we completely encapsulate the TCP/IP-layer. All operations executed above the DHT layer use only functionality of the DHT. Handling of TCP/IP connections and related issues are due to the DHT. The DHT is responsible for managing the peers, i.e., it handles joining peers, leaving peers, churn, etc. Additionally, it provides mechanisms for handling lookups and message routing.

We introduced DHT systems in Section 3.2.2. Concerning the challenges presented in Section 2.1, DHTs reveal the following advantages:

- they provide the foundation for achieving scalability in terms of participants, data amounts, and query processing
- the guarantees concerning message and hop complexity are fundamental prerequisites for realizing efficient cost-based query processing at large scale
- mechanisms for automatized self-organization and maintenance algorithms for highly dynamic systems are included
- achieving robustness and high availability in unreliable networks is a main feature of these systems
- fairness and efficiency are based on a solid ground, due to the (implicit or explicit) load balancing in these systems
- certain aspects of privacy come "for free", because there exists no global knowledge at all

Due to using only standard DHT functionality, we do not restrict ourselves to any specific system. Nevertheless, in parts we utilize special functionality that is not provided by all systems by default. At the corresponding places, we highlight why and for what exact purpose such special features are used. However, these are only options for leveraging efficiency. The basic query processing functionality proposed in this work can be implemented on top of any standard DHT system. The following list summarizes the features that are desirable, but not mandatory:

- support of efficient range queries
- support of prefix queries
- load-balancing features

Note that assuming these features is not unrealistic. Several DHT systems meanwhile support these and other sophisticated capabilities. The motivation is to integrate novel processing paradigms and efficient query processing techniques already on DHT level. Recently, some DHT systems support even multi-attribute indexing and querying. There are several systems providing all three of the capabilities listed above. Meanwhile, almost every popular DHT system supports efficient processing of range queries, either by nature [60,16,29] or by extension [182,80,130]. Queried ranges are mapped to according key ranges and these key ranges are looked up accordingly. Thus, range query mechanisms can be applied to every data indexed in the system. They can be used to implement specific operators exceptionally efficient, see Chapter 7. For systems not supporting range queries we can at least provide rather simple implementations using standard DHT functionality. The idea is to route to the first peer of the range and afterwards to all peers of the range in a sequence.

procedure call	**meaning**
`first-key(`p`)`	determine the smallest possible hash key with prefix p
`last-key(`p`)`	determine the largest possible hash key with prefix p
`next-key(`p_1, p_2`)`	determine the next possible hash key with prefix p_2 following the largest possible hash key with prefix p_1
`forward(`k`)`	forward a message to the peer responsible for hash key k
`forward(first-key(`p`))`	forward a message to the first peer responsible for prefix p, i.e., responsible for the smallest possible key with prefix p
`range-forward(`p`)`	forward separate messages to all peers responsible for prefix p (using a prefix or a range query)
`range-forward(`min, max`)`	forward separate messages to all peers responsible for the range from hash key min to hash key max

Table 4.1: DHT procedure calls used in this work

P-Grid is a DHT that natively supports prefix queries, due to its prefix-preserving hashing. Other systems regard prefix queries not as important as range queries, but often they can be mapped to range queries. Prefix queries can be used for special operator implementations as well. Load balancing is either implicitly provided (due to uniform hash functions) or explicitly aspired in DHTs. There is no query processing strategy or operator implementation that urgently requires this feature. But, it is beneficial for achieving good query performance, due to the fair distribution of data and query load. We utilize it also for determining guarantees like the completeness of query answers, see Chapter 7. In the algorithms presented throughout this work, we use several procedure calls that refer to DHT functionality. Table 4.1 summarizes these procedure calls and their meaning. If not stated otherwise, we will expect prefix queries to be sent to all peers responsible for the given prefix, not to only one of all.

4.1.1 The P-Grid P2P System

This work proposes an architecture that can make use of any available DHT overlay. Nevertheless, we base explanations on the P-Grid DHT [1,3]. There are several reasons for that. First of all, a running implementation of P-Grid is available and the developers can be contacted for specific questions. This enabled us to build a prototype (see Chapter 8) that can make use of the full DHT functionality. Further, P-Grid supports all

requirements of this work, such as efficient prefix and range querying, sophisticated load-balancing techniques, and the preservation of data correlations due to order-preserving hashing. In the following, we give a brief overview of the main characteristics of the P-Grid overlay system.

Traditional DHTs use uniform hashing functions to map application keys to their identifier space. While this achieves good storage load-balancing and efficient discovery of exact keys, it is in conflict with preserving the semantic proximity. Existing relations among the application-specific keys are destroyed and keys that are semantically close at the application level are heavily fragmented in the DHT. P-Grid organizes nodes in a binary trie structure, which is a standard indexing structure from databases to support, among others, range queries. The word "trie" is a combination of "tree" and "retrieval", indicating the popularity of this ordered data structure for retrieval tasks. The keys assigned to the nodes of a trie are usually strings s. The position of a node n in the tree encodes the corresponding key $s(n)$: The root is associated with an empty string and all descendants of a node n_p share the same common prefix $s(n_p)$. Edges are annotated with substrings s_i, such that $s(n_i) = s(n_p)||s_i$ for all direct children n_i of n_p, i.e., for all edges $n_p \rightarrow n_i$. $||$ denotes string concatenation. In a binary trie, each node except the leafs has two outgoing edges. A key benefit of tries is that they cluster semantically close data items.

Without constraining general applicability, P-Grid uses binary keys. A generalization to k-ary structures is discussed in [11]. Keys are computed using an order-preserving hash function. Each edge is either annotated by "0" or "1". Thus, the keys corresponding to a node on level l are binary strings of length l (the root is located at level $l = 0$). Each peer p is associated with a leaf of the binary trie, which identifies the key space partition that the peer is responsible for. Thus, the trie exists only virtually and is not completely materialized by according peers. It is constructed by only pair-wise interactions without any central coordination or global knowledge. The binary string Π_p that corresponds to the leaf's key is called the peer's *path*. A peer is responsible for all index items with a hash key prefixed by its path Π_p. Each peer p stores a set of index items I. Π_p is a prefix of or equal to the hash key $h(I)$ for all locally stored I. It is not excluded that temporarily other index items are also stored. P-Grid supports load balancing for nearly arbitrary skewed data [7]. Following, each peer in P-Grid is responsible for approximately the same number of index items. This results in imbalances in the trie, i.e., single subtrees can theoretically become arbitrarily deep. For fault-tolerance, query load-balancing, and hot-spot handling, multiple peers are associated with the same key-space partition (structural replication) and peers maintain multiple references to other peers with the same path (data replication). Consistency between replicas is assured by epidemic algorithms. Updates in P-Grid are driven with only lose consistency guarantees [59].

Lookups that cannot be answered locally have to be forwarded towards the responsible peer. P-Grid implements a prefix-based greedy query routing. For this, a peer maintains a set of known peers in a routing table. The routing table is constructed to hold peers with exponentially increasing distance in the key space from the local peer's position. This technique basically builds a small-world graph [115], which enables search in a logarithmic number of steps. Let Π_p^l denote the prefix of Π_p with length l, i.e., the key for p's ancestor at level l. For each prefix $\Pi_p^l, 0 < l \leq |\Pi_p|$, a peer stores a set of references

Π_q to peers q with property $\overline{\Pi_p^l} = \Pi_q^l$. $\overline{\Pi}$ is the binary string Π with the last bit inverted. This means, Π_q^l has exactly the same prefix of length $l-1$ as Π_p^l and differs only in the last bit. Like this, the peer has references to some other peers at each level of the trie. These references represent links to other parts of the trie – the longer the common prefix, the shorter is the range of the link.

For looking up a key k, a peer p determines the longest common prefix Π_p^l between Π_p and k. If $l = |\Pi_p|$, peer p can answer the lookup locally. Otherwise, a peer q is chosen from level $l+1$ of the routing table and the lookup is forwarded to q. Π_q has at least one bit more in common with k. This routes the lookup towards the requested index item(s), because q is closer in the key space to k than p. The process is repeated until the responsible peer is hit. The algorithm always terminates, as long as the P-Grid trie is complete and a peer from each partition is reachable. The first is guaranteed by P-Grid's construction algorithm, the second is achieved by applying replication. Due to P-Grids small-world characteristics, lookups are guaranteed to finish in $O(\log N)$ steps, where N is the number of unique key space partitions (i.e., the number of peers divided by the structural replication factor). Thus, $O(\log N)$ messages and $O(\log N)$ hops are guaranteed. [2] shows that this holds for arbitrary shapes of the trie, i.e., for arbitrary data distributions. Routing tables can require up to $O(N)$ space for arbitrarily skewed data. In practice, they are usually by far smaller, because they are bounded by the depth of the underlying trie. For successful lookups of arbitrary keys, at least one peer has to be known for each level of the routing table. Note that these are by far less references than key space partitions exist. To increase robustness and query-load distribution, multiple peers are stored for one level in the routing table. During lookups, according peers are chosen randomly.

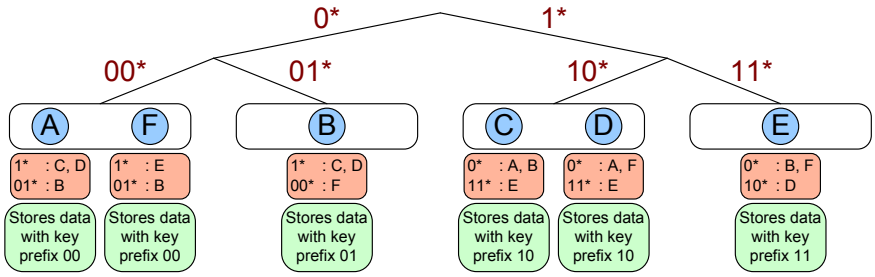


Figure 4.1: P-Grid overlay network

Figure 4.1 shows a simple example of a P-Grid tree. It consists of 6 peers responsible for 4 partitions. Peer F's path is 00. Thus, F is responsible for all data with key prefix 00. The length of F's path results in two entries in its routing table: peer E with path 11 at the first level and peer B with path 01 at the second level. A search initiated at peer F for key 100 would first be forwarded to peer E, because it is the only entry in F's routing table at level 1*. As peer E is responsible for 11 and not for the key 100, peer E further forwards the query to peer D, which can finally answer the query.

We assume that P-Grid ensures hash keys that are long enough. This means, there exists no index item with a key that is a prefix of any peer's path. Otherwise, the index item could be stored at any of the peers with such a path. Note that in this case there would be at least two peers with a prefixed path. This assumption is important only for

overhead considerations. A query for a key k that is too short would result in a prefix query, which is forwarded to all peers with that prefix. Thus, all according index items would be found.

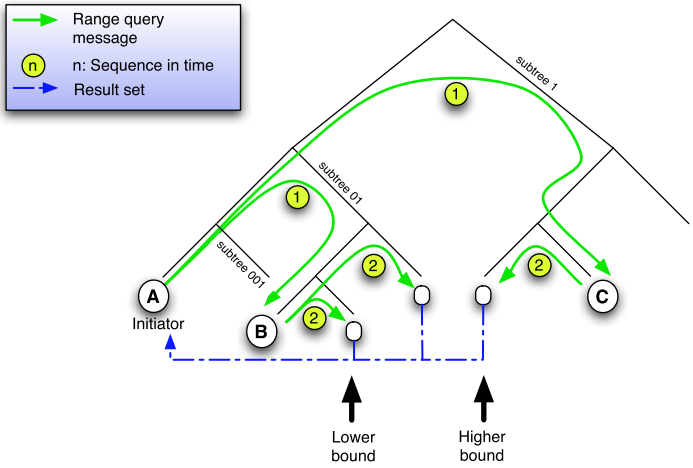


Figure 4.2: Range query illustration

Range queries in P-Grid are first forwarded to an arbitrary peer q responsible for any of the key space partitions within the range. Afterwards, the query is forwarded to the other partitions in the interval using q's routing table. The process is recursive. Since the query is split into multiple queries that appear to trickle down to all the key-space partitions in the range, it is called the *shower algorithm* [60]. The intuition of the algorithm is illustrated in Figure 4.2. A range query from 0101 to 1000 is initiated at peer A. As peer A's path is 000, it forwards the query to subtree 01 and subtree 1 in parallel. According references in the routing table may point to peer B with path 0100, peer C with path 101, respectively. Both peers are still not in the queried range and forward the query accordingly. From its own path, peer B again identifies two relevant subtrees, which are 0101 and 011. Peer C only forwards to path 100. The contacted peers are in the range and can answer the query. Further, they recognize that there are no more peers in their subtree and can finish processing. During forwarding, it is possible that the query is forwarded to a peer responsible for keys outside the range. However, it is guaranteed that this peer will forward the query to a key-space partition within the range. The P-Grid routing ensures that no key space partition will get duplicates of the range query.

[60] also proposes a more intuitive algorithm that could be applied to any structured overlay. The basic idea is to first find the lower (or upper) bound of a range and then sequentially forward the query along neighbors till the upper (or lower) bound is reached. This simple approach has the disadvantages that *(i)* a lost query message results in the immediate termination of the range query as no further peer is reached and no further data will be returned; and *(ii)* forwarding one message sequentially along neighbors can result in a very long query response time depending on the size of the queried range and the number of peers involved. Therefore, the shower algorithm should be the preferred choice. Its performance is considerable better in terms of response times and robustness, but it requires only a slightly larger number of messages. [188] provides a corresponding discussion on the costs of the shower algorithm.

4.2 Data Model

We suggest to use a triple-based storage model closely related to the idea of RDF [171]. Similar to RDF, a triple t consists of the following three parts:

1. An identifier, called object ID (OID or *oid* for short) – this can be a URI, a system generated ID, etc. The OID is used to identify triples that belong to each other.
2. A property, also called predicate or attribute (a).
3. An attribute value v, belonging to the predicate.

We only apply the triple scheme. We do not rely on the graph describing semantics as RDF does. Thus, query processing differs essentially. The proposed triple model offers several advantages. It is:

- self-descriptive: a triple "tells" to which attribute of which tuple it refers and all attributes belonging to one tuple can be looked up (thus, there is no need for a global data dictionary)
- extensible: new tuples following an arbitrary schema can be added at any time and existing tuples can be modified by adding, removing, or modifying existing triples
- universal: data from different application domains can be mixed and enriched by metadata without altering the triple model

Several types of (semi-)structured data can be mapped to this triple scheme. Mapping RDF data is straightforward, because RDF is also modeled as triples. Following, the modeling of all data types that can be represented in RDF is intuitive as well.

In the next sections, we describe how raw user data, such as relational data and XML data, and simple semantic metadata can be mapped to this triple model. This highlights that we achieved what we have been aiming for: a generic and flexible schema for representing heterogeneous user data. Further, we show that semantic correspondences can be modeled in the same way as raw user data are. This provides a basis for tightly integrating such correspondences into query processing. For the remainder of this work, we exemplary focus on the usage of relational data enriched by semantic mapping data.

4.2.1 Indexing Relational User Data

Relational data is modeled as tuples that conform to a given schema. Inserted tuples are not managed in their original form. They exist only virtually. Each tuple is split into triples and these triples are inserted into the DHT. We refer to original tuples using *ot*. The set of triples resulting from all inserted *ot* is referenced by *OT*. Imagine a tuple *ot* with schema:

a_1	a_2	...	a_n
v_1	v_2	...	v_n

$a_i, i \leq n \in N$ are the attribute names of the tuple. Let $v_i, i \leq n$, refer to the values corresponding to the attributes a_i. Such a tuple is split into triples as follows:

oid_t	a_1	v_1
oid_t	a_2	v_2
	...	
oid_t	a_n	v_n

oid_t is the object ID that identifies all triples belonging to the original tuple ot. This ID may be system generated. Alternatively, an identifying value like a primary key may be used. Like this, a tuple consisting of n attributes is split into n triples.

As mentioned before, the underlying DHT implements a horizontal data partitioning approach. By adhering to the triple format, we create a vertical partitioning of the input data. When these triples are inserted into the DHT and distributed, we combine the horizontal and vertical partitioning schemes. This is the aforementioned combination of the universal relation model [135] and column stores [197].

A triple is inserted into the DHT using a hash key k. This key is determined by applying a hash function h on the triple. A crucial question is on which part(s) of the triple h should be applied. We answer this question below. The combination of a triple and its corresponding hash key is called an index item. Index items can be looked up by searching for the corresponding hash key k. All triples indexed with the same hash key are located on the same peer. Triples with succeeding hash keys are located on neighboring peers (with respect to the topology of the DHT). By inserting triples multiple times, each time using a different hash key, we can build different indexes on top of the data. This allows for providing different access paths to the data managed in the system. Which and how many indexes are built depends on the application requirements. It represents a trade-off between performance of query processing and storage overhead. Building I indexes on top of a tuple with n attributes results in $I \cdot n$ triples that are injected into the system. Thus, if we insert m tuples, each having $n_j, j \leq m$, attributes, we end in $\sum_{j=1}^{m} I \cdot n_j$ triples.

Useful indexes on triples of the form $(oid\ a\ v)$ are:

1. An OID index using $h(oid)$, for efficiently querying all triples that correspond to one original tuple ot (materializing the tuple)

2. An attribute-value index on the concatenation of a and v ($h(a||v)$), for processing queries containing predicates like $a\theta c$, where θ refers to an operation like $<, =, \geq$, etc. and c to a constant that is compared to the values of a (i.e., to v)

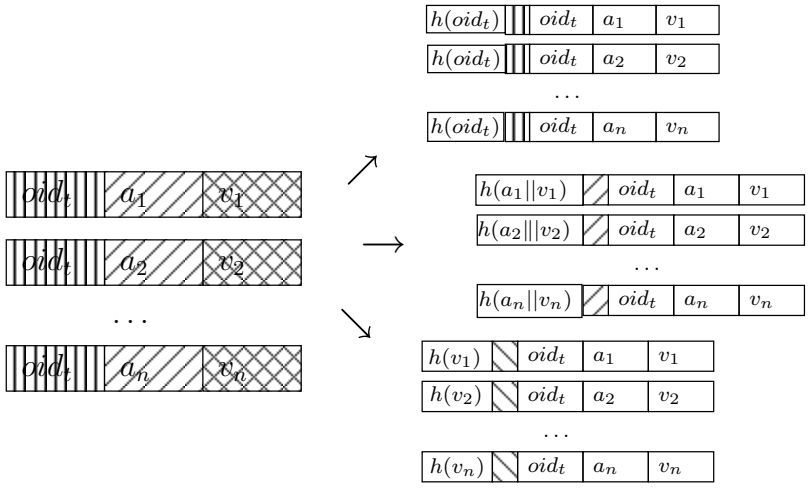
3. A value index using $h(v)$, for querying arbitrary values without referencing a specific attribute

The usefulness of index 2 is leveraged if the underlying DHT supports prefix queries. Then, a lookup operation for $h(a)$ supports materializing all triples for attribute a – without the need of an additional index. Note that a pure attribute index would be problematic. As all corresponding triples would be hashed to the same key, all index

items for one attribute would be stored at only one peer. As in traditional databases, the indexes can be build on only a selection of attributes, OIDs, or values. This influences the availability of physical operators, which are presented in Chapter 6. We only have to assure that each triple is indexed at least one time – or we will not be able to ever find it again.

The set of indexes can be extended for supporting expensive operations like similarity queries. Note that the actual number of generated index items depends on the type of index. In the examples above, each triple results in exactly one index item for each index. Alternatively, h may be determined on, for instance, substrings of an attribute name. Then, the number of created index items depends on the number of indexed substrings. We present such an index for specifically supporting string similarity in Section 6.2.1.

Summarizing, the following illustration shows how the three indexes described above are built by assigning a different fill pattern to each indexed part:



In P-Grid, we have to use order-preserving hash functions for indexing the triples. For strings, we apply a hash function that preserves the prefix relationship $\subseteq$ between strings. This means,

$$s_1 \subseteq s_2 \rightarrow h(s_1) \subseteq h(s_2) \quad .$$

For numerical data, we apply a hash function that preserves the natural order of values, i.e., based on the well-known Euclidean distance. The reasons for these choices are clarified throughout this work. Fortunately, P-Grid cares about all load-balancing issues that result from the used hash functions. We should further ensure that not too many index items are mapped to the same hash key. In this case, no load-balancing approach can work. An option is to add suffixes to very frequent hash keys for indexing and to use prefix queries on the original keys for querying. Figure 4.3 shows an example of two tuples indexed in P-Grid using the OID index (`OID`$\rightarrow$), the attribute-value index (`A`$||$`v`$\rightarrow$), and the value index (`v`$\rightarrow$). The two tuples result in 18 triples distributed over the peers while respecting order-preservation. The OIDs are system generated.

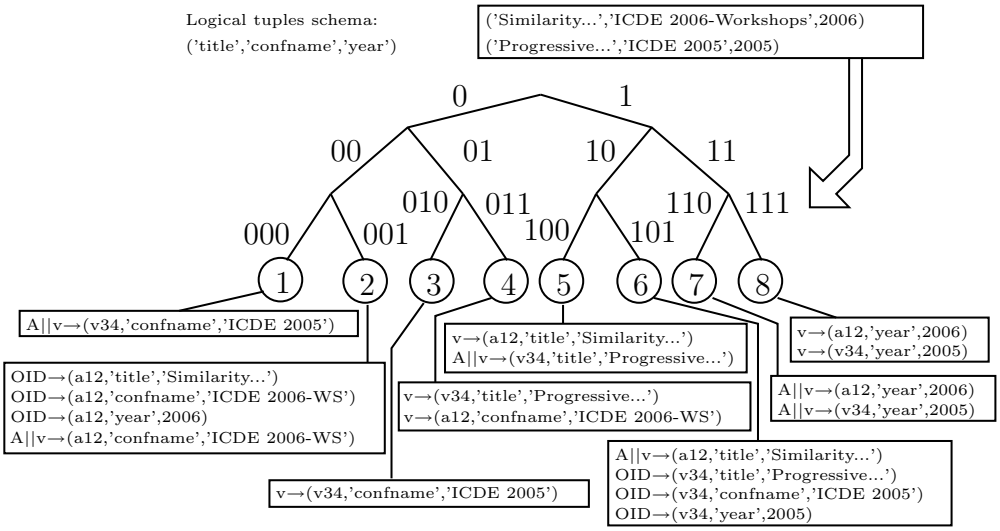


Figure 4.3: Triples in the P-Grid overlay network

4.2.2 Indexing XML User Data

We use XML data as a second example of user data, because it is very popular in a wide range of Web-based applications. Public Data Management will strongly benefit from the support of XML data.

There has been much work on how to map XML data to relational data. If this can be accomplished, mapping it to the triple model is straightforward following the instruction from Section 4.2.1. There are also several works discussing how to map XML data to RDF data [139, 23]. As RDF data is very close to our triple model, we base the following explanations on these works.

There are two general choices to map XML data to RDF. Either use generic 'has-child' relationships between parent and child or a more informative approach by naming the relationship appropriately. [139] follows the first approach. The authors suggest to use element names to classify the content of the elements and to map attributes to the value part of triples. System-generated IDs are used for that. The core concept of [23] is to map every XML element and attribute to the value part of a triple. This models the XML structure as a relational model between parent nodes and their children. The approach makes use of knowledge from accompanying XML schema definitions. This results in using element names in all parts of triples. Below, we exemplarily show how to map a small XML document to the triple model based on [139]. The resulting triple data is more intuitive and it presents a generic mapping between XML and the triple model. For increasing readability, we omit the usage of the typical XML namespaces and use only core element names. The following is a snippet from an XML document:

```
<car>
  <mileage greaterThan="5000">5324</mileage>   ,
  <price>10000</price>
  <info>
    <p>Great car, <em>amazing</em>!</p>
  </info>
  <info>
    <p>see <a href="pic.jpg">picture</a></p>
  </info>
  <dealer>
    <name>Mike</name>
    <addr>
      <city>Boston</city>
      <street>15th Av</street>
    </addr>
  </dealer>
</car>
```

It can be modeled by the following set of triples:

```
(genid-1, type, document)

(genid-1, car, genid-2)

(genid-2, mileage, genid-3)
(genid-2, price, 10000)
(genid-2, info, genid-4)
(genid-2, info, genid-5)
(genid-2, dealer, genid-6)
(genid-2, value, ",")

(genid-3, greaterThan, 5000)
(genid-3, value, 5324)

(genid-4, p, genid-7)

(genid-7, em, amazing)
(genid-7, _1, "Great car,")
(genid-7, _2, !)

(genid-5, p, genid-8)

(genid-8, a, genid-9)
(genid-8, value, see)

(genid-9, value, picture)
(genid-9, href, pic.jpg)

(genid-6, addr, genid-10)
(genid-6, name, Mike)

(genid-10, street, "15th Av")
(genid-10, city, Boston)
```

Imbalance problems stemming from frequent triple attributes like `value` are handled by P-Grid. Here again, it is important to use the attribute-value index rather than an index on only attribute level. If such clustered attributes shall be avoided, an approach similar to [23] should be preferred. Further, it might speedup typical queries if XML paths like `car/mileage` or `car/dealer/./city` are indexed rather than single element names. This section is only ought to clarify that XML data can be managed straightforward. We do not focus on special issues from XML P2P systems [118].

4.2.3 Representing Semantic Correspondences

Before, we motivated the exigent need of support semantic metadata. These data are needed in order to describe relations between different domains, schemata, real-world objects, etc. combined in the universal data store. Preferably, this metadata should fit into the proposed data model straightforward. This is needed to successfully split the data model and the query layer from the semantic level. Only then semantic query expansion and handling of mappings can be encapsulated in a separate layer.

We already mentioned that RDF data can be represented in the triple model without modifications. RDF data is a standard in the Semantic Web community and thus, the triple model implicitly supports the representation of semantic correlations. Nevertheless, in many situations rather simple mapping information will be sufficient. In this case, modeling such information in a few triples accompanying the raw user triples should be supported on a simplified basis without the need of a full-fledged semantic model like RDF. In this section, we suggest an according approach with reduced complexity.

The basis for achieving semantic interoperability is formed by pair-wise schema correspondences. These correspondences build the mapping graph. We start our considerations from the fundamental correspondence type *attribute equivalence* $A \cong B$. This correspondence type expresses that attribute A of schema one represents the same information as attribute B in schema two. This basic relation can be found very often in the considered scenarios. For instance, one user or service supporting attribute "birth", a second one providing the same information in attribute "dob". This is illustrated in a small example about city data in Figure 4.4. Two relations are sketched and two equivalence mappings between attributes are indicated. This can be enriched by other correspondence types like *subclass*. Even correspondences defined on whole relations are considerable.

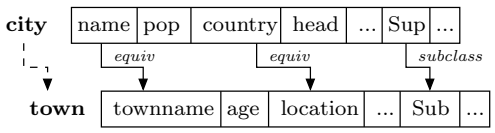


Figure 4.4: Example schema correspondences

Such mapping information can be represented in the proposed storage model using a single triple, e.g., `(country map:equiv location)`. More relations can be modeled using different identifiers in the predicate part of the triple. Also, additional information can be stored. So, a triple `(A map:crtdate 2007-12-03)` may be used to represent the creation date of the mapping. Even more complex mappings like `person.name` $\Leftarrow$ `p.firstname||p.lastname` can be described by a set of simple triples.

Of course, correspondences between attributes may be much more complex, spanning from algebraic operations (e.g., currency translation) over attribute combinations (concatenation, splitting, ...) to complex filter statements (e.g., when defining complex views). We focus on basic correspondences, enriched by the optional involvement of additional data like ontology information. An advantage of applying the triple model is that metadata can be integrated easily and well-fitting into query formulation and processing. No extended knowledge and no special techniques are needed for that. Semantic triples are indexed and managed exactly as raw data triples are. Like this, the mapping graph information is distributed over the participating peers. The (automatic) integration of correspondences into query formulation is introduced in Section 5.1.2. Their integration into query processing is matter of Section 5.1.2.

There has been much work on representing ontologies in a triple format, such as RDF. Several of these approaches qualify for representing ontology information in our framework. One popular approach used in many applications is OWL [160]. The actual expressiveness and applicability of the used form of representation is not in the focus of this work. OWL can be expressed in RDF. Thus, it can be used in the triple store without restrictions. Throughout this work, we restrict ourselves to some basic, intuitive, and self-explanatory relations, such as *subclass*, *superclass*, etc. An exemplary representation for a set of transitive relations is listed below. Attribute `A` belongs to an ontology concept `C`, which is a subclass of concept `P`. `P` itself is a subclass of concept `G`. A simple set of triples can represent this:

```
(A ont:belongs C) (C ont:domain ...)
(C ont:subclass P) (P ont:subclass G)
...
```

This can be easily extended to relations between multiple ontologies and to vocabularies like OWL. Similar, an attribute is not bound to only one ontology or only one relation. A triple like `(A ont:belongs C)` describes that attribute `A` belongs to an ontology class `C`. Then, on class `C` more relations can be modeled by further triples. Thus, `(A ont:belongs C)` represents a kind of entry point into complex ontologies.

The approach reveals several advantages:

- additional mapping information can be added with ease
- easy integration into storage and processing model
- information management and discovery is distributed

Of course, this comes along with increased query overhead (the mapping information must be queried as well) and increased storage overhead. We regard these costs as implicit and unavoidable costs of realizing semantic interoperability.

In SQL views can be used for defining arbitrary schema mappings. Correspondence definitions formulated in XML, XQuery, or similar [84] are popular as well. We briefly pick up the idea of views for representing mappings in Section 5.1.1, after introducing the needed basics of the used query language. But, this approach requires special functionalities and extended semantics. We believe that a rather simple set of mapping types, which can be represented using simple triples, will be satisfactory in most situations.

Moreover, if these basic correspondences can be combined with the navigation in an arbitrary shaped ontology, which is expressed by a set of simple triples as well.

4.3 Related Work

A couple of commercial and open source cloud computing projects were already introduced in Section 3.1. We do not discuss them here as related work. They grew up in the field of distributed systems research and do not adopt a database view as we do. Currently, they are rather used as simple storage solutions for sharing multimedia documents and managing backups. Nevertheless, architectural approaches like using a DHT on the physical layer can be found also in these approaches. To the best of our knowledge, [35] is the first work discussing data management overlays on top of storage services like S3. This combines database functionality with *utility computing* [178] in a way as we propose to combine it with P2P overlays. The authors focus on traditional database aspects, such as the trade-off between consistency and availability, particularly in the presence of small objects and frequent updates. They discuss the opportunities and limitations of this approach in the context of general-purpose database applications. Query processing is part of their considerations, but not discussed as recondite as in our work. Further, we believe in the higher degree of distribution provided by P2P systems. Google also started a project aiming at the support of database features like indexes and schemata on top of SimpleDB [1]. It will be interesting to see the next outcomes of this new trend in the field of database research.

First works considering complex database functionality on top of P2P overlays are [78] and [27]. Both works propose first visions on the combination of the two research fields. While [78] focuses on general opportunities and challenges and discusses the problem of data placement, [27] deals with problems of data integration.

A very closely related work that is also motivated by database requirements for P2P systems is PIER [100]. The authors also suggest to use a DHT overlay on the physical layer. PIER lacks functionality to handle heterogeneous data, which is mandatory in Public Data Management scenarios. PIER supports querying homogeneous Internet data, i.e., there is a globally known data schema. On the storage layer, fragments of relations are distributed over the participating peers following the idea of horizontal data partitioning. Tuples are annotated by relation name, attribute names, and attribute types. A selected partition attribute serves for building a primary index. In addition, secondary indexes that contain only references are supported. The tuples are stored only once in the primary index. Consequently, PIER supports standard SQL queries. The system implements a soft-state storage, which means data expires after a certain time and needs to be refreshed. Similar to the work in hand the focus of PIER is on query processing. We discuss PIER's query processing techniques and strategies in more detail in Section 6.6. Similar goals are aimed by PeerDB [156, 153], which is based on the DHT BestPeer [152]. But, PeerDB supports only full-fledged database systems as sources. These sources are accompanied by centralized directory servers. Thus, we regard that approach as a first step towards Public Data Management that bears disadvantages concerning the scalability in terms of data and participants.

[1] http://perspectives.mvdirona.com/2008/07/10/GoogleMegastore.aspx

RDFPeers [41] is a distributed infrastructure for managing large-scale sets of RDF data. Due to the triple model of RDF, the storage layer and data model of RDFPeers are very similar to our work (see Section 4.2). RDFPeers is based on the multi-attribute addressable network MAAN [42]. Each part of a triple is indexed, but whole triples are stored each time. Our approach using P-Grid is very similar, because we also store complete triples, but index only parts of them. Numerical data are hashed using a locality-preserving hash function. Load-balancing is discussed as well. There exist several other proposals for large-scale distributed RDF repositories [87, 198] and for managing and querying RDF metadata [151]. [87], for instance, maps the whole RDF model graph to nodes of a DHT. Further, RDF-Schema data are indexed in a distributed manner. These RDF graphs are combined during query processing to match query graphs. All of the works apply a vertical partitioning scheme on the set of triples. We combine this approach with the idea of the universal relation model [135] and aspects from column-oriented databases [197]. The resulting triple model is introduced in Section 4.2. All the RDF stores index different parts of triples in parallel, but none of the works discusses different indexing schemes as comprehensive as we do. The universal relation model is popular since the early 80s for simplifying data integration. It has been successfully adopted in a multitude of works. But, concerning P2P data management, this idea can be found in only few proposals. For instance, the group of Ooi plans to use their "one table stores all" solution [157] in conjunction with BestPeer [152]. Also, some of the utility platforms described before are based on the idea of a universal relation [48, 164]. The vertical approach relieves us from deciding on the best fragmentation and distribution scheme for horizontal partitioning. This decision usually depends on the kind of queries to expect and the kind of input data [185].

[14] provides a comprehensive overview of content distribution technologies in P2P networks, including DHT-based approaches. But, the work surveys according approaches from a network point of view and loses sight of database-related issues. Due to the lack of analyzing database-like data modeling approaches, query processing is based on rather simple lookups. The close relationship between content distribution and query processing approaches demands a closer look on according models. Nevertheless, it gives valuable insights into existing systems and also discusses security-related issues. Further, [14] briefly discusses the differences between grid computing and P2P technologies. This is also provided by [155]. A discussion of issues concerning XML P2P data management, including indexing, clustering, replication, and query processing can be found in [118].

Part III

Query Layer

Chapter 5

Query Language

Up to here, we introduced the distribution layer of the proposed architecture and the applied triple model. The distribution layer provides a basis for realizing large-scale distributed data management and querying. The triple model provides a basis for mightier data models, schema correspondences act as a basis for mightier mappings. Before we can get over to discuss query processing approaches, we introduce a query language as an interface to the focus of this work. The introduced logical algebra represents the interface between queries and physical execution plans by mapping the semantics of the language to logical operators.

In Section 5.1, we introduce the used query language and the underlying logical algebra. As RDF was invented for modeling and representing (Semantic) Web data, SPARQL [165] was developed for querying and filtering this data. Thus, we base the used query language on the syntax and ideas of SPARQL. We introduce ways for integrating semantic correspondences into query formulation in Section 5.1.2.

We focus on the algebra, because the semantics of the operators defined here are crucial for the physical implementations discussed in Chapter 6. The actual language and its syntax is introduced informally with the help of examples. We abstain from providing a comprehensive formal definition due to its affinity to well-known query languages. We rather provide a convenient basis for understanding the subsequent discussions than distract from the essentials by introducing too much details. Details about the language and a parser implementation can be found in [190]. Related work in Section 5.2 summarizes the proposals that inspired our approach.

5.1 Vertical Query Language and Algebra

This section introduces the proposed query language. There are several requirements that such a language should fulfill. Most of them are common for query languages in general, but some specifically apply to the proposed scenarios. The language should be:

- designed for the vertical triple model
- declarative, intuitive, and familiar to a large community
- based on the principle of logical data independence, i.e., querying has to be independent from the location of data
- set-oriented
- suited to formalize advanced query constructs like similarity and ranking
- implicitly supporting queries on schema level

Other requirements, such as closeness, completeness, appropriateness, orthogonality, etc., apply in general for query languages. It is intuitive to base an according language on well-known and popular existing approaches. On the one hand, the proposed *Vertical Query Language* (VQL) is a version of SPARQL [165] providing a subset of the features. SPARQL was designed for the RDF triple model and is very popular in the Semantic Web community. On the other hand, it extends SPARQL features by supporting special constructs like similarity and ranking. As SPARQL took its cue from SQL [47] concerning syntax and semantics, VQL does as well. We assume the reader's familiarity with the basic concepts of these query languages. We introduce the basic notion using some small examples. The focus of this section is on the underlying logical operators and their semantics, which provides the basis for succeeding discussions on physical implementations of these operators. We use database-like notations for representing query plans and the contained operators:

- a query plan is illustrated as a tree
- the vertices of such a tree represent the query operators
- edges between vertices define the flow of data between operators
- query plans are processed from the down to the top – the output of the root operator is the final query result

This section is structured as follows:

1. We give insight into the basic syntax of VQL.
2. We introduce further formalisms concerning the triple model, which are required for understanding the explanation of the algebra.
3. We define the basic algebra operators and their semantics. The process of transforming VQL queries into logical query plans and subsequent optimization steps are overviewed as well.
4. Finally, we illustrate language and algebra by means of examples.

Basic Syntax The basic syntax of SQL follows the well-known `SELECT-FROM-WHERE` construct. In our case, the combination of triple model and universal relation supersedes the `FROM` clause, because there are no separated relations, documents, etc. We handle only one universal (but virtual) relation. This is already reflected by the basic SPARQL syntax. In SPARQL, a `FROM` clause is optional, binding the processed triples to only those collections listed in this clause. For VQL, we can generally restrict the basic syntax to a `SELECT-WHERE` construct. The basic syntax of VQL is defined using the Extended Backus-Naur Form in Appendix A. In this section, we present it on an example basis and focus on the underlying algebra. Afterwards, we illustrate language and algebra in combination using more complex examples. The following is a simple VQL query:

```
select ?name
where { ?o 'name' ?name}
```

VQL supports the notion of variables, which are used in the triple patterns that are contained in the `WHERE` clause. Variables are identified by a leading question mark. The structure of result rows is specified by the list of projected variables in the `SELECT` clause. The query from above returns all values of triples with the attribute `name`.

The `WHERE` clause contains the triple patterns that triples of the result set have to match. Triple patterns are separated by a dot and are combined by conjunction. If the keyword `distinct` is provided in the `SELECT` clause, duplicate result rows will be filtered out. This duplicate detection is processed after the projection: rows are generated as defined by the projection and duplicate detection is processed on only the resulting rows of values. For instance, the following query returns all distinct combinations of values for `name` with corresponding values for `addr`:

```
select distinct ?name, ?addr
where { ?o 'name' ?name . ?o 'addr' ?addr}
```

The usage of the same OID variable `?o` results in a join between triples on the value of this variable. This means, each result row consists of two values extracted from a set of triples that belong together. Each of these triple sets consists of two triples that belong to the same originally inserted tuple *ot* – identified by an equal OID. Each variable has to be bound unambiguously. For non-atomic attributes (i.e., multiple existing triples with the same OID `?o` and attribute name `addr`, but different values `?addr`) this results in multiple result rows, one for each triple with the according attribute name.

The special projection `?var.*` projects all attributes belonging to the same OID – only variables referencing OIDs are allowed here. This is very useful to produce result rows that contain all available attribute values, i.e., reconstructing original tuples *ot*. Thus, to rebuild all original tuples that contain an attribute `name` we can use the query:

```
select ?o.*
where { ?o 'name' ?name}
```

This combines all corresponding triples on their OIDs and finally projects the values that are stored in the third part of these triples. `?o*.*` can be used to return pairs of attribute names (second triple part) and values (third triple part), `?o*` can be used to return only the attribute names.

Triples can also be combined despite different OIDs. This corresponds to a standard join in the relational model. The join condition is provided by according variables. The following query returns joined values of `name` and `addr` under the assumption of two join attributes `info` and `id`:

```
select ?name, ?addr
where { ?o 'name' ?name . ?p 'addr' ?addr .
        ?o 'info' ?info . ?p 'id' ?info }
```

As shown, the triple patterns may contain constants to explicitly bind parts of the regarded triples. String constants are embraced by '. All terms that are not keywords, not preceded by a question mark, and not embraced by ' are regarded as numerical constants. Bindings can be defined also implicitly using `FILTER` statements. The following query corresponds to the last query from above:

```
select ?name, ?addr
where { ?o ?a ?name . ?p 'addr' ?addr .
        ?o 'info' ?info . ?p 'id' ?id
        filter ?a='name' filter ?info=?id}
```

Filter expressions define additional conditions. Multiple expressions are combined by conjunction. Without loss of generality, we assume atomic predicates in each filter expression. Disjunctions can be expressed by according `union` statements. A query that returns all values of attributes named `name` or `title` looks like:

```
select ?v
where { ?o ?n ?v filter ?n='name'
       union
        ?o ?t ?v filter ?t='title'}
```

In this case, all projected variables have to be bound in both parts of the union. Logical `AND` and `OR` relations could also be integrated, as well as nested unions and set operations, such as `IN`, `ALL`, and `ONE`. We focus on the fundamental functionalities introduced so far. In contrast to SPARQL, all filter expressions are listed after all triple patterns. They cannot be mixed up. This is just for implementation reasons [190] and makes no restriction. A couple of functions can be used in the filter expressions. For instance, the *edist* function computes the edit distance (or Levenshtein distance [126]) between two strings. The following query receives all values for attributes named similar to `name`. The values are listed together with the attribute names. This illustrates how handling the schema level is supported through variables:

```
select ?a, ?name
where { ?o ?a ?name filter edist(?a,'name')<2 }
```

The basic syntax is enriched by constructs for ordering result elements (`ORDER BY`), computing skylines (`SKYLINE`), a limiting clause (`LIMIT`), etc. We illustrate them and the underlying operators using small examples below.

Triple Model The suggested data model is introduced in Section 4.2. Here, we introduce some more formalisms that describe how triples are handled during query processing. Based on this, we define the semantics of query operators.

As introduced, a triple t is defined as:

$$t := (oid\ a\ v)$$

Intuitively, this is the input for each operator on leaf level of a query plan, because all data managed in the system are stored in triple format. The three parts of a triple t are referenced by applying an array-like notation: $t[oid]$ refers to the OID (triple identifier), $t[a]$ to the attribute name (triple property), and $t[v]$ to the attribute value (triple value). During query processing, different triples are extracted from the DHT and combined according to the query plan. The result is a set of triple sets. Each triple set $ts := \{t\}$ can be compared to a result tuple of SQL queries. This means, all triples of one ts provide the values for one single result row. A final projection produces that set of result rows following a schema defined by the query, similar to relational queries that produce new result relations. At the leaf level of query plans, the triple sets consist of only one triple, i.e., $|ts| = 1$. During query processing, triples are combined to form larger triple sets.

One way to extend triple sets is to *materialize* additional attributes belonging to the same originally inserted tuple ot. Triples that belong to each other can be identified by the OID that each of them contains. Thus, materialization can be achieved by combining triples having the same OID, which refers to an equi-join on the OIDs.

Further, triple sets may be combined using a join. Consequently, each ts can contain triples with different OIDs. Each ts can be stored in the system in the same way as we store original tuples ot. For this, we just have to create a new OID that refers to the triple set. This new OID can be used later to identify all triples that comprise the triple set. Each operator takes one or two sets T of ts as input and produces a set T of ts as output (except the projection operator, see below). Details follow below when we describe the semantics of the corresponding operators. Note that this data model supports a closed algebra. The result of any operator can be materialized and stored in the system without breaking the triple model. Further, the output of any operator can be used as input for any other operator.

The algebra is based on a set semantic. We use set operations $|\ldots|, \cup, \setminus$ and $\cap$ for handling triple sets. This means that the triples contained in a triple set are unique. Further, no duplicate triple sets are contained in the final result. For this, we have to define the equivalence relations:

$$t_1 = t_2 \Leftrightarrow t_1[oid] = t_2[oid] \wedge t_1[a] = t_2[a] \wedge t_1[v] = t_2[v]$$

$$ts_1 = ts_2 \Leftrightarrow ts_1 \setminus ts_2 = \emptyset \wedge ts_2 \setminus ts_1 = \emptyset$$

An important property is the duplicate-free character:

$$\forall ts \in RT(\forall t_1 \in ts \forall t_2 \in ts(t_1 \neq t_2)) \wedge \forall ts_1 \in RT \forall ts_2 \in RT(ts_1 \neq ts_2) \quad ,$$

where RT refers to the set of triple sets composing the final query result. This is an important requirement, because identical triples may be combined with the same triple set multiple times. This is due to inserting one triple several times using different indexes

(and different hash keys to distinguish the DHT index items). We will clarify this later on. The set semantic prevents results from blowing up unnecessarily by duplicates. Duplicate result triple sets due to independent parallel processing are prevented as well. Duplicate result rows can still occur, because only parts of a subset of all triples in a triple set may be projected. To prevent this, the `distinct` keyword can be used in the `SELECT` clause of VQL queries.

Next, we introduce the operators of the logical algebra. We also indicate basic rules for transforming VQL queries into logical query plans. The set of rules is not exhaustive, but provides a basic understanding of the relations between query language and query algebra. More details about the VQL syntax (in an earlier version) and about mapping triple patterns and filter expressions to query operators can be found in [190]. The author describes a parser implementation for VQL.

Query Operators As mentioned above, the only operator that does not directly produce triple sets ts as defined by the triple model is the projection operator. We handle this operator first. The projection operator refers to the `SELECT` clause of a VQL query and determines the final result, which is composed of variable values. This is different to the relational algebra, where the result is composed of attribute values. The variable values are projected from all ts of the set RT of final triple sets. This leverages query expressiveness and flexibility, because any element of a triple (identifier, schema, or instance level) can be part of a result row. For instance, this allows for issuing:

- pure schema queries
- queries containing similarity predicates on schema level (e.g., is attribute `Titel` similar to attribute `title`!?), allowing that the original attribute names may also be part of a result row

The produced result rows could be modeled as triple sets as well. We abstain from discussing that. Consequently, we consider the projection operator always placed at the root of a query plan. We refer to the mapping from a variable $?var$ to the corresponding part of a triple from one $ts \in RT$ as $ref(?var, ts)$. This information is maintained and shipped with the intermediate results. In the following, we focus on the actual triple sets and omit the references $ref()$. We only refer to them at the relevant places.

π: *Projection.* The final result set of a query is defined as a set of value sets. Each contained set makes one row and comprises all projected variables $?var$ and their values. The relations between variables and values are indicated by $\rightarrow$. The projection operator π is defined on RT as:

$$\pi_{\texttt{List}}(RT) := \{ \bigcup_{?var \in \text{'}\texttt{List}\text{'}} \{?var \rightarrow ref(?var, ts)\} | ts \in RT\}$$

The following indicates the general rule for transforming the `SELECT` part of a VQL query into an according projection operator:

The operators presented in the following are defined as either $T \rightarrow T$ (unary operators) or $T \times T \rightarrow T$ (binary operators).

In order to handle triple data, some special operators are required. In the following operators, we use so-called *variable-triple patterns* (VTPs) to model the relationship between variables and corresponding triple parts. We introduce them first.

VTP: *Variable-Triple Pattern.* A VTP consists of a variable pattern on the left side and a triple pattern on the right side of an arrow. The different parts of each pattern are separated by "·":

$$\texttt{VTP} := [?o{\cdot}?a{\cdot}?v \rightarrow \texttt{o} \cdot \texttt{a} \cdot \texttt{v}]$$

`o, a,` and `v` are either number or string constants (strings are embraced by '). A triple matching a VTP has values as specified in the triple pattern. Further, variable references are set as specified in the variable pattern.

All parts of both patterns are optional. If no parameters are provided in the triple pattern, the pattern may be omitted at all:

$$[?o{\cdot}?a{\cdot}?v \rightarrow] \hat{=} [?o{\cdot}?a{\cdot}?v \rightarrow \cdots]$$

If no variable is bound, the variable pattern and the arrow may be omitted:

$$[\texttt{o} \cdot \texttt{a} \cdot \texttt{v}] \hat{=} [\cdots \rightarrow \texttt{o} \cdot \texttt{a} \cdot \texttt{v}]$$

A common way of using VTPs is to extract a list of attributes and their values. Often, only the extracted attribute values are referenced in the π operator. For convenience, we use a shorter notation for this:

$$[\text{'}\texttt{attr}\text{'}] \hat{=} [\cdot\cdot?attr \rightarrow \cdot\text{'}\texttt{attr}\text{'}\cdot]$$

Several triple patterns from a VQL query can be mapped directly to VTPs. The following illustrates some example rules:

```
select ?name
where { ... ?o 'name' ?name ... }
```
$\Rightarrow$ ['name']

```
where { ... ?o ?a ?name ...
        filter ?a='name' ... }
```
$\Rightarrow [?o{\cdot}?a{\cdot}?name \rightarrow \cdot\text{'}\texttt{name}\text{'}\cdot]$

```
where { ... ?o ?a 123 ...
        filter ?o='xyz' ... }
```
$\Rightarrow [?o{\cdot}?a{\cdot} \rightarrow \text{'}\texttt{xyz}\text{'}\cdot\cdot\texttt{123}]$

VTPs are only used in the following two operators, which are the only operators where variable references can be set.

ξ: *Extraction.* First, we need an operator ξ that extracts triples at the leaf level of query plans. Though triples can be modeled as triple sets and thus, ξ could be defined on T, we define it on OT. This clarifies the way of extracting triples on the leaf level of query plans. We define the extract semantic for the case that only the attribute name is provided as follows:

$$\xi_{[\text{'attr'}]}(OT) = \xi_{[\cdot ?attr \rightarrow \text{'attr'} \cdot]}(OT) := \{\{t\} | t \in OT \wedge t[a] = \text{'attr'}\}$$

The value of attribute `attr` is referenced by variable $?attr$. We store the following references together with the (intermediate) result:

$$ref(?attr, ts) = t[v]$$

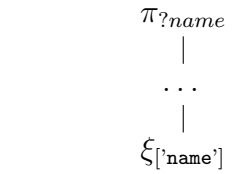
The operation extracts triples t that conform to the triple pattern and transforms them into triple sets ts. We omit the definition for other cases of the VTP as well as a general definition, because they are straightforward. If parts other than the attribute name are provided, the condition $t[a] =$ 'attr' will change accordingly. If multiple parts are provided, the corresponding conditions will be combined by conjunction. In each case, all variable references are set accordingly. The set of references is maintained and passed along with intermediate results up to the end of query processing.

An empty triple pattern corresponds to extracting all triples stored in the DHT. This results from VQL queries that contain triple patterns in which each variable is unbound. Intuitively, this is expected to be an extremely expensive operation. We only support this on logical level in order to transform VQL queries into query plans. For physical processing, we reject any queries that contain triple patterns with unbound variables. Note that a binding can result from other triple patterns, as we will show below.

As shown above, we can extract a set of VTPs from a VQL query. The query planner has to choose one VTP for each OID variable used in the query. These VTPs are mapped to ξ operators at leaf level. Thus, two more example rules for plan generation are:

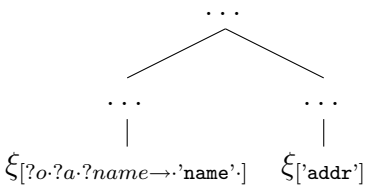
```
select ?name
where { ... ?o 'name' ?name ... }
```

⇒




```
where { ... ?o ?a ?name ...
        ?p 'addr' ?addr ...
        filter ?a='name' ... }
```

⇒



ω: *Materialization.* Similar to ξ, we need an operator that adds triples to existing triple sets above the leaf level of query plans. ω allows for materializing such triples. This is done on the basis of identical OIDs. Similar to ξ, the operator materializes all triples conforming to a triple pattern. The OIDs on which to combine triples with triple sets are provided by a reference variable $?r$. For this, we introduce a further information that triple sets are annotated with: $oid(?r, ts)$ returns the OID of the triple that $?r$ refers to in ts. Actually, a variable refers to one part of a triple. But, due to a smart implementation, both methods $ref()$ and $oid()$ are supported using only one small data structure. See Section 8.1 for details. As for $ref()$, we only refer to $oid()$ at the relevant places and focus on the actual triple sets otherwise. ω needs to access T as well as OT, though it is a unary operator. Its intended primary use is to materialize additional attributes for already materialized triple sets. This reconstructs (parts of) original tuples ot. Thus, we show the definition for this case:

$$\omega_{?r;\texttt{VTP}}(T) := \omega_{?r;\texttt{VTP}}(T, OT)$$

$$\omega_{?r;[\texttt{'attr'}]}(T, OT) := \{ts \cup \{t\} | ts \in T \wedge t \in OT \wedge t[a] = \texttt{'attr'} \wedge t[oid] = oid(?r, ts)\}$$

Additionally, corresponding references are set:

$$ref(?attr, ts) = t[v]$$

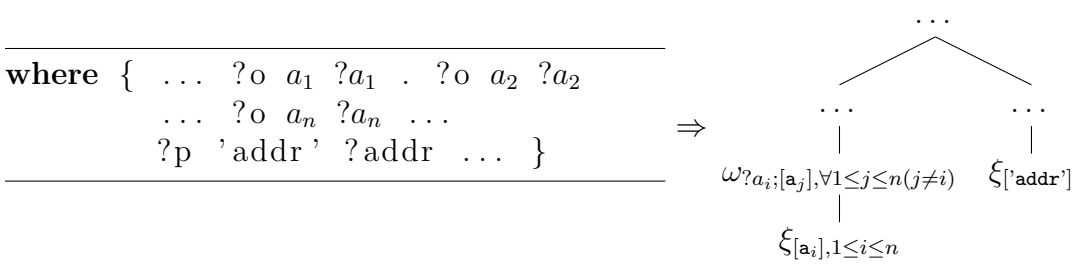
The materialize operator ω is very similar to the extraction operator ξ. Thus, the definitions for other cases of VTPs follow the same rules. From a conceptual view, the ω operator corresponds to a ξ followed by a join on the OIDs.

The single variable-triple pattern may be replaced by a list. Then, an accommodated triple conforms to any of the provided triple patterns. Like this, multiple triples are merged into the triple set:

$$\omega_{?r;[\texttt{'a1'}],[\texttt{'a2'}],...,[\texttt{'aX'}]}(T, OT) \hat{=} \omega_{?r;[\cdot?a1 \rightarrow \texttt{'a1'}\cdot],[\cdot?a2 \rightarrow \texttt{'a2'}\cdot],...,[\cdot?aX \rightarrow \texttt{'aX'}\cdot]}(T, OT)$$

The operator follows a kind of inner join semantic: If any attribute from the list is not found for an input triple set ts, ts will be filtered out. Of course, this holds for all cases of VTPs. Note that there is always a triple $t' \in T$ with $t' = oid(?r, ts)$, because only variables that are already bound are allowed for reference. Otherwise, $oid(?r, ts)$ would not return a valid OID. This semantic can be modified if needed. Also, a kind of outer materialize operator can be added. ω also supports to materialize all triples that exist for an OID using an asterisk sign: $\omega_{?r;*}$. This special operation demands a special implementation, which we discuss later.

All VTPs that are not mapped to a ξ operator have to be mapped to according ω operators. They have to be placed in the subtree that also contains the ξ operator for the same OID. The following example illustrates this rule:



Such a combination of extract and materialize operators is used for arbitrary VTPs, as long as the referenced OID is unique.

Next, we introduce operators known from the relational algebra.

σ: *Selection.* In analogy to the relational algebra, filter expressions that are not directly mapped to VTPs are usually mapped to selection operators σ:

$$\sigma_{cond}(T) := \{ts \in T | cond(ts) = true\}$$

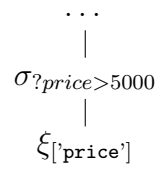
$cond(ts)$ represents the condition that is evaluated on each input triple set ts. For this, the condition contains variable references. All variables have to be bound, which can be checked during query parsing. Thus, each input triple set ts contains a valid reference for each used variable $?var$. For plan generation, the atomic predicate from each `FILTER` statement is mapped to a σ operator. The supported types of atomic predicates follow from the definition of VQL:

- $?var$ *op* c, where c is a constant and $op \in \{=, <, \leq, >, \geq, \neq\}$
- $func(?var, c)$ *op* c, which refers to a function like $edist()$ evaluated on variable $?var$ and constant c
- $?var_1$ *op* $?var_2$, with $op \in \{=, <, \leq, >, \geq, \neq\}$ – this compares two variables $?var_1$ and $?var_2$
- in analogy, $func(?var_1, ?var_2)$ *op* c is supported

The following illustrates the general rule for inserting σ operators into query plans:

```
where { ... ?o 'price' ?price ...
        filter ?price>5000 ... }
```

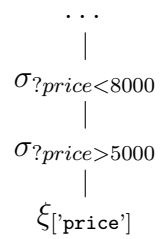
⇒



In this manner, all filter expressions that contain only one variable and that are not transformed directly into VTPs are transformed into a combination of σ and ξ or σ and ω. Multiple filter expressions on the same variable are first represented as separate selections:

```
where { ... ?o 'price' ?price ...
        filter ?price>5000 ...
        filter ?price<8000 ... }
```

⇒



Multiple successive selections can be combined during query optimization. Similar to VQL, disjunctions and set operations like `ALL` and `IN` could also be integrated into the algebra. This can be helpful for combining multiple selections during the process of logical query optimization.

Note that ξ can be seen as a selection σ on the set OT. Two equivalent algebra terms are:

$$\xi_{[\cdot ?a \cdot ?attr \rightarrow \cdot \texttt{'attr'} \cdot]}(OT) \hat{=} \sigma_{?a=\texttt{'attr'}}(\xi_{[\cdot ?a \cdot ?attr \rightarrow]}(OT))$$

Similarity operations combined with extractions are handled in analogy. In logical query plans they are always represented similar to:

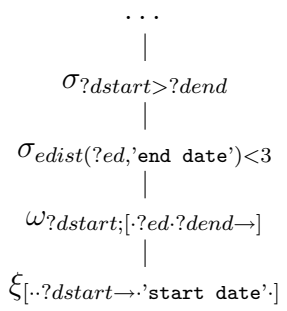
$$\sigma_{dist(?a,\texttt{'title'})<3}(\xi_{[\cdot ?a \cdot \rightarrow]}(OT))$$

The condition in σ specifies a binding for the variables in the VTP. In physical query plans this is replaced by one operator combining both logical operators. Like this, we prevent looking up all index items stored in the DHT. See Chapter 6 for more details about this. In analogy, similarity predicates for materializations are represented by combining σ with ω accordingly.

The comparison of two variables requires that the variables are bound in the input triple sets. Usually, this is used to combine and filter two independent triple sets on the provided condition. But, also two attributes referring to the same OID can be compared. For example:

```
where { ... ?o 'start date' ?dstart
        ... ?o ?ed ?dend ...
        filter ?dstart>?dend ...
        filter edist(?ed,'end date')<3
        ... }
```

$\Rightarrow$



For applying a selection on triple sets that have been combined despite different OIDs the next operators can be used.

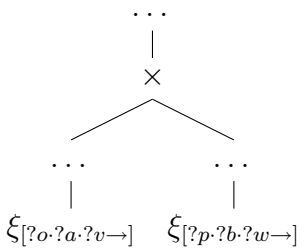
$\times, \bowtie$: *Cross-product* and *Join*. Two triple sets are combined using the $\times$ operator:

$$\times(T_1, T_2) := \{ts_1 \cup ts_2 | ts_1 \in T_1 \wedge ts_2 \in T_2\}$$

Remember that this is duplicate-free, i.e., a triple that is contained in ts_1 and ts_2 will be contained only once in the resulting ts.

A cross-product is indicated by two triple patterns that share no expression and are not related by a filter expression. The rule for generating $\times$ operators looks like the following:

where { ... ?o ?a ?v ...
?p ?b ?w ... }



Again, this is a general rule that applies also to other forms of triple patterns, i.e., arbitrary VTPs in the extract and materialize operators preceding the cross-product. If triple patterns with different OIDs share a variable or are related by a filter expression, $\times$ can be combined with a selection σ. This represents a join under a condition. Similar to the relational algebra, we define several $\bowtie$ operators as a combination of $\times$ and σ:

– $\bowtie_{cond}$ joins triple sets according to a condition *cond*:

$$\bowtie_{cond} (T_1, T_2) := \sigma_{cond}(\times(T_1, T_2))$$

– $\bowtie$ defines a natural join on triple sets:

$$\bowtie (T_1, T_2) :=\bowtie_{ncond} (T_1, T_2)$$

$$ncond := (\exists t_1 \in ts \exists t_2 \in ts(t_1[oid] \neq t_2[oid] \wedge t_1[a] = t_2[a]))$$

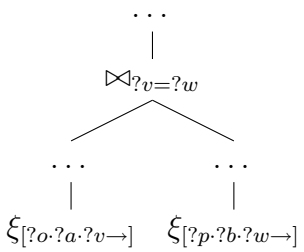
$$\wedge$$

$$(\forall t_1 \in ts \forall t_2 \in ts(t_1[oid] \neq t_2[oid] \wedge t_1[a] = t_2[a] \rightarrow t_1[v] = t_2[v]))$$

Each triple set *ts* referenced in the condition is one output triple set from the cross product. This rejects triple sets without at least one common attribute (in the first part of *ncond*). The second part describes that all equally named attributes must have equal values in order to be part of the result. This shows a strength of the triple model: operations on schema and instance level can be arbitrarily combined – they only differ in the part of a triple that is actually processed.

An according rule looks like:

where { ... ?o ?a ?v ...
?p ?b ?w ...
filter ?v=?w ... }



$\cup$: *Union.* The union operator simply combines the input sets of triple sets. In contrast to $\times$ and $\bowtie$, existing triple sets are not modified nor filtered out.

$$\cup(T_1, T_2) := \{ts | ts \in T_1 \cup T_2\}$$

Other set operations like $\cap$ and $\setminus$ are defined analogously. We omit any transformation rules, because intuitively they are directly indicated by the corresponding language keywords.

$*$: Most other operators known from the relational algebra can be integrated without modifying their semantics, except adapting them to the triple model. Up to this point in time, we have integrated a renaming operator β and a grouping operator γ. The more important one is the γ operator, which we primarily intend to use for determining simple aggregate values.

All of the introduced operators that are based on some condition ($\sigma, \bowtie$) support similarity predicates. The similarity between values is determined using a distance measure, which is implemented in appropriate functions like *edist* for the edit distance between strings (see Section 6.2.1 for details). The notion of variables allows for defining arbitrary similarity predicates on identifier, schema, and instance level. This is exceptionally useful in heterogeneous environments, where semantically equivalent concepts may be described by literally similar attribute names. Additionally, the probability of spelling errors, different but similar abbreviations, etc. rises significantly in Public Data Management scenarios. This can be handled using similarity operations as well. The similarity join conforms to the definition from [55]: Given two input sets of tuples r and s with schemas $r : (A_1 \dots A_k)$ and $s : (B_1 \dots B_l)$ and values $r : (a_1 \dots a_k)$ and $s : (b_1 \dots b_l)$, a similarity join produces the cross product of all tuples and returns tuples with schema $(A_1 \dots A_k B_1 \dots B_l)$ for which a similarity predicate $edist(a_i, b_j) < c$ ($i \leq k \wedge j \leq l \wedge c$ is constant) is true. We adopt this notion to triple sets and extend it to the schema level and to a combination of both, schema level and instance level.

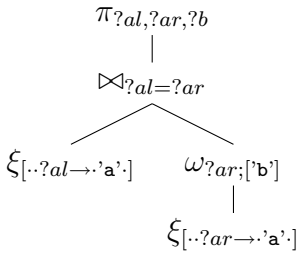
First Example As data model and used data structures may be difficult to grab abstractly, we give a small example before completing the list of operators. This example lends itself to be placed here, because we can illustrate the special triple operators ξ and ω as well as join semantic and set semantic in the special case of a self-join. On the first look, a self-join as known from the relational world seems to be impossible using the vertical algebra. This example illustrates that this is not the case. It is intended to familiarize the reader with the triple model – examples of VQL queries and corresponding algebra representations follow later. Imagine a simple and small relation containing only two attributes `a` and `b`. Two rows of this relation would be transformed to the triple model as follows:

	a	**b**
x	1	8
y	1	9

$\Rightarrow$

$$\begin{array}{llr} (x & \text{a} & 1) \\ (x & \text{b} & 8) \\ (y & \text{a} & 1) \\ (y & \text{b} & 9) \end{array}$$

x and y are system-generated OIDs. Now, imagine the following query in algebra notation:



Attribute `a` is extracted at leaf level of both branches and later used as join attribute. Thus, this represents a self-join. If this query is executed against the 4 triples listed above, the intermediate result sets will look as follows. Both operators at leaf level generate the following equal triple sets:

left branch				right branch			
ID	ts	$ref(?al, ts)$	$oid(?al, ts)$	ID	ts	$ref(?ar, ts)$	$oid(?ar, ts)$
1	(x a 1)	1	x	3	(x a 1)	1	x
2	(y a 1)	1	y	4	(y a 1)	1	y

In the right branch, attribute `b` is materialized in addition, which extends the right input of the join operator as follows:

left branch				right branch					
ID	ts	$ref(?al, ts)$	$oid(?al, ts)$	ID	ts	$ref(\ldots, ts)$		$oid(\ldots, ts)$	
						?ar	?b	?ar	?b
1	(x a 1)	1	x	3	(x a 1), (x b 8)	1	8	x	x
2	(y a 1)	1	y	4	(y a 1), (y b 9)	1	9	y	y

Afterwards, the self-join is processed and results in the following triple sets:

ID	ts	$ref(\ldots, ts)$			$oid(\ldots, ts)$		
		?al	?ar	b	?al	?ar	b
1×3	(x a 1), (x b 8)	1	1	8	x	x	x
1×4	(x a 1), (y a 1), (y b 9)	1	1	9	x	y	y
2×3	(y a 1), (x a 1), (x b 8)	1	1	8	y	x	x
2×4	(y a 1), (y b 9)	1	1	9	y	y	y

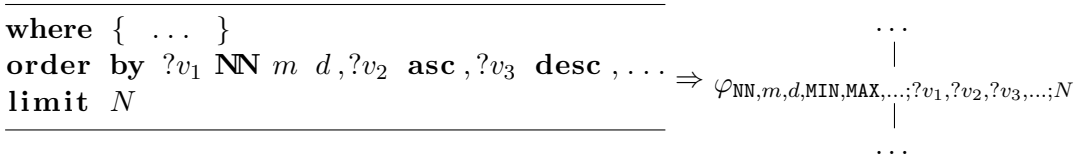
The resulting tuples seem to have a different "structure". This is only for minimizing the size of triple sets in the result set. All variable references set before are kept and thus, $ref()$ and $oid()$ can still be evaluated correctly. In the end, the result of the projection looks as expected:

?al	**?ar**	**?b**
1	1	8
1	1	9
1	1	8
1	1	9

Up to here, we introduced all operators needed to meaningfully represent structured queries on top of the proposed triple model. The notion of variables and the integral support of similarity operations leverage the usefulness in heterogeneous data collections. There are other important operators tailor-made for large-scale data management and querying. These rank-aware operators are very useful for IR-related tasks. Due to the complexity of these operations, we introduce them only informally. Supported operators are:

φ: *Top-N*. The operator $\varphi_{rank;\texttt{List};N}$ sorts the set of input triple sets according to the k ranking functions from *rank*, which are evaluated on the k variables from `List` in the given order. Optionally (if $N > 0$), the resulting set is pruned to the size N. This corresponds to a top-N query. Supported ranking functions are: `MAX`, `MIN`, and `NN,m,d`. `NN` stands for Nearest Neighbor ranking: the result contains all/the N closest items in a maximal distance of d from m. For pruning either $d \geq 0$, $N \geq 0$, or both can be specified. If $d \leq 0 \wedge N \leq 0$, the result will be ordered according to the distances to m, but it will not be pruned. Ranking functions on strings apply the edit distance by default, for numerical data we support the Euclidean distance as default. A multidimensional top-N query can be defined by providing $k > 1$ ranking functions and variables. Generalizing the operator for arbitrary distance measurements and further ranking functions is straightforward. For instance, ranking functions on multiple attributes could be supported by allowing new variable definitions, such as $?r = 0.5 \cdot ?price + ?dist$ and including $?r$ in `List`. The functionality as presented above shall be sufficient for discussing the operator implementation(s) in detail.

The usage of a top-N operator is indicated by the `ORDER BY` and `LIMIT` clauses of a VQL query:



Φ: *Skyline*. In extremely large data collections it is essential to support ranking of query results. Often, multiple attributes have to be included into the ranking process. Multidimensional top-N queries are very helpful, but it can be tough to decide about the priority of the ranking goals. It can be very beneficial if a result represents the set of best possible trade-offs between all ranking goals. The database world knows this as *skyline* queries [33]. Skyline queries are, for instance, extremely helpful in decision making tasks. A very popular example is to query for all hotels that are preferably close to the beach (`distance MIN`) *and* preferably cheap (`price MIN`). A well-known definition of skylines is based on the *dominance* relation [33]: given two or more ranking functions, the skyline of a dataset consists of all points that are not dominated by any other point. Thus, a member of the skyline is ranked higher than or equal to all other members in at least one dimension. Further, a member is ranked higher than any non-member in at least one dimension of the skyline. In mathematics and economy this set is known as the *Pareto optimum*. We introduce more details on this concept in Section 6.2.2. $\Phi_{rank;\texttt{List}}$ represents the skyline operator. *rank* contains one ranking function for each attribute from `List`. Thus, $|rank| = |\texttt{List}| \geq 2$.

VQL supports a special `ORDER BY` construct to formulate skyline queries:

where { ... }
order by skyline of
  $?v_1$ **MIN**, ... , $?v_k$ **MAX**

$\Rightarrow$

$$\vdots \;\; \Phi_{\texttt{MIN},\ldots,\texttt{MAX};?v_1,\ldots,?v_n} \;\; \vdots$$

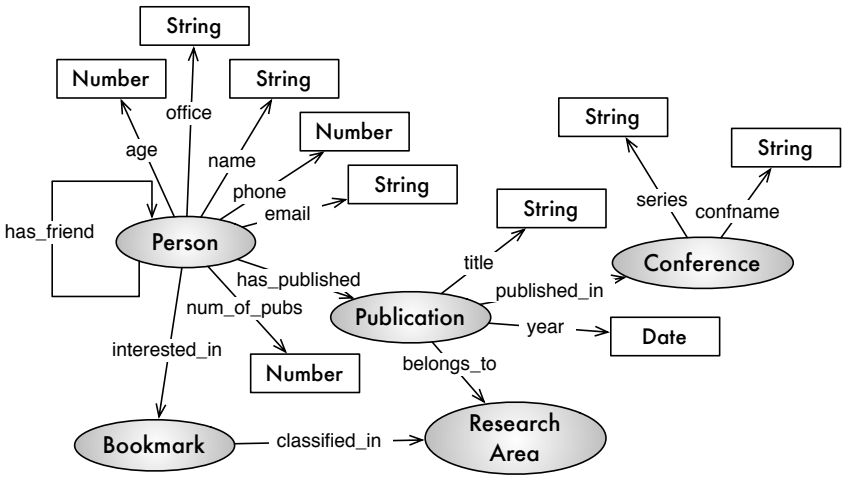


Figure 5.1: Example schema

An illustrating example beside the popular hotel scenario is based on the schema sketched in Figure 5.1. The schema models data about authors and their publications as it could be expected in a Public Data Management scenario. The figure shows several entities and their attributes, as well as relations between them. Analyzing the data to identify correlations between the age of authors and the number of publications can result in interesting insights. A corresponding query could use the `MIN` ranking in the age attribute and `MAX` in the number of publications. But, none of the two ranking goals should be prioritized, which results in a skyline query. Assuming that each entity is modeled as a separate relation and then mapped to the triple model, an example VQL query is the following.

```
select ?name,?age,?cnt
where { ?a 'name' ?name . ?a 'age' ?age . ?a 'num_of_pubs' ?cnt .
        ?a 'has_published' ?title . ?p 'title' ?title
        ?p 'published_in' ?conf . ?c 'confname' ?conf .
        ?c 'series' ?sr filter edist(?sr,'ICDE')<2
      }
order by skyline of ?age MIN, ?cnt MAX
```

The resulting skyline reaches from the youngest authors to those authors published the most publications. Note that skyline queries become even more powerful in conjunction with advanced querying features like similarity queries. In the example, we are looking for only authors published at ICDE workshops – considering possible data heterogeneities the query allows an edit distance of up to 1 to the term 'ICDE' in order to ignore typos and similar.

Ranking operators are usually placed near to the root of a query plan. The main requirement is that they are located above all operators that they depend on, e.g., the according extract and materialize operators.

operator	symbol	meaning
Projection	π	project result rows
Extraction	ξ	extract triples at leaf level
Materialization	ω	materialize triples at higher levels
Selection	σ	filter triple sets
Cross-product	$\times$	combine triple sets without a condition
Join	$\bowtie$	combine triple sets under a condition
Union	$\cup$	merge triple sets
Grouping	γ	group triple sets
Top-N	φ	rank triple sets
Skyline	Φ	rank triple sets using skyline semantic
Map	μ	resolve schema mappings

Table 5.1: Logical query plan operators

Table 5.1 summarizes all logical query plan operators that have been introduced so far. The map operator μ is introduced in Section 5.1.2. Next, we sketch the complete process of query transformation and logical optimization.

Overview of Plan Generation and Logical Optimization During the introduction of the vertical algebra we also indicated rules for transforming VQL constructs into logical query plans. Further, we mentioned options that a succeeding logical plan optimization offers. In this section, we provide a brief overview of the complete process. The main steps of the process of transformation and logical optimization are:

1. Parse the VQL query.
2. Check the query for syntactical correctness.
3. Check the query for semantical correctness (as far as possible). This includes a parameter check for function calls, the correct and unambiguous usage of constants and variables, binding of variables, etc.
4. Simplify the query by removing redundant patterns and expressions. Further, expressions are simplified.
5. Produce a first plan as indicated by the presented transformation rules. This involves the mapping of implicit and explicit conditions to variables and OIDs, the derivation of extractions and materializations, the identification of joins, etc.
6. Afterwards, the process of logical optimization starts on the basis of the first query plan. Obvious rules are to move selections downwards and to push materializations upwards. But, the optimizer has to consider that materializations can prevent ambiguous results and unwanted self-joins due to the "outer" semantic (see the examples below). Further, the VTPs of ξ and ω operators can be exchanged and operators can be combined. Also, the identification of shared expressions is a prerequisite for efficient query processing.

The optimized query plan provides the input for the succeeding physical optimization. Details of that process are introduced in Section 6.4.

Examples In the following, we illustrate the usage of VQL queries on the basis of some examples. This is meant to highlight special features and to familiarize with the functionality of VQL as a basis for further discussions on query processing. For each query, we present one corresponding algebra term as a logical query plan. This is meant to illustrate the transformation of VQL queries into algebra terms and the used operators.

As a simple example, consider information about cars and dealers, which are represented by two relations with typical attributes such as name, mileage, price, etc. Let us assume the following original relations:

- `dealers`: (`id, name, addr`)
- `cars`: (`name, hp, price, mil, dlr`) with the referential constraint `dlr` → `dealers.id`

The tuples of these relations are transformed into triples and indexed accordingly. We start with some basic queries. The following queries imply knowledge of the structure of the inserted relational data. This is perfectly suited for illustrating VQL and the underlying algebra. In the end, we expand this to queries without (complete) knowledge about data schemata.

Imagine a user who wants to know about the name, horsepower (hp), price, and mileage (mil) of each car having more than 250 hp, ordered by hp. He could issue the following query:

```
select ?name,?hp,?price,?mil
where { ?o 'name' ?name . ?o 'hp' ?hp . ?o 'price' ?price .
        ?o 'mil' ?mil filter ?hp > 250}
order by ?hp desc
```

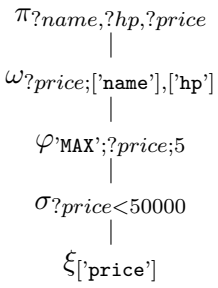
This corresponds to a logical query plan as follows:

$$\pi_{?name,?hp,?price,?mil}$$
$$\omega_{?hp;['\texttt{name}'],['\texttt{price}'],['\texttt{mil}']}$$
$$\varphi_{\texttt{MAX};?hp;0}$$
$$\sigma_{?hp>250}$$
$$\xi_{['\texttt{hp}']}$$

Concerning extremely large data sets, a good idea might be to restrict the result set to only the five most powered cars. The following query exactly achieves this. Further, the result set is restricted to cars with a maximal price of 50.000. This represents a top-N query:

```
select ?name,?hp,?price
where { ?o 'name' ?name . ?o 'hp' ?hp . ?o 'price' ?price
        filter ?price < 50000 }
order by ?hp desc limit 5
```

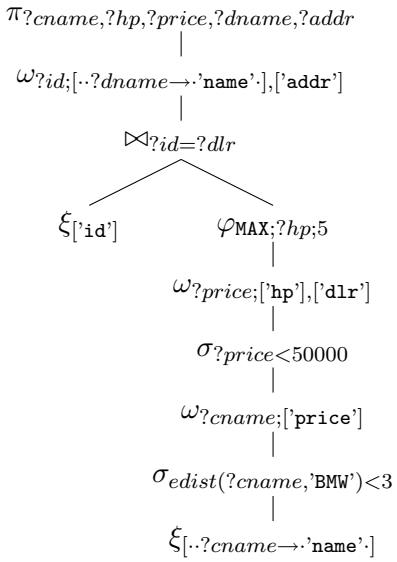
A corresponding query plan is:



Now, we can enrich the result set by additionally querying for all corresponding dealers and their addresses (addr), which involves join operations. However, we are only interested in BMW cars:

```
select ?cname,?hp,?price,?dname,?addr
where { ?x 'dlr' ?d . ?y 'id' ?id .
        ?x 'name' ?cname . ?x 'hp' ?hp . ?x 'price' ?price .
        ?y 'addr' ?addr . ?y 'name' ?dname
        filter ?price < 50000 filter edist(?cname, 'BMW') < 3}
order by ?hp desc limit 5
```

Here is a corresponding query plan:

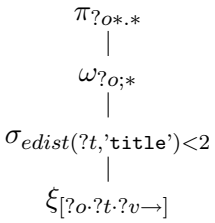


Note that $\xi_{[\cdot\cdot?cname\rightarrow\cdot\texttt{'name'}\cdot]}$ will also extract triples belonging to the relation `dealers`, because attribute `name` exists in both original relations. However, these triples will be filtered out when trying to materialize attribute `price`, because this attribute does not exist in `dealers`. A more convenient way to handle this is to use prefixes in the attribute names. This helps to identify the original relations, e.g., using `dealers:name` and `cars:name`. The methods for query formulation and processing proposed in this work remain unchanged in this case.

So far we included some "traditional" operations as well as ranking and similarity on instance level. A powerful aspect of the triple store is the possibility to express similarity on schema level, which simplifies integration tasks. The following query returns all attribute names and values of originally inserted tuples that include an attribute similar to `title`:

```
select ?o*.*
where { ?o ?t ?v filter edist(?t,'title') < 2}
```

This query shows that it is straightforward to query for information about data schemata. The projection `?o*.*` lists the attribute names and their values of all result triples. This is the only way to materialize triples without explicitly referencing them in the `WHERE` clause. Resulting query plans look like:



This example highlights that we have to support similarity operations on the extraction level. Otherwise, we would have to materialize all triples that are managed in the system. In Section 6.2.1 we present according physical operators.

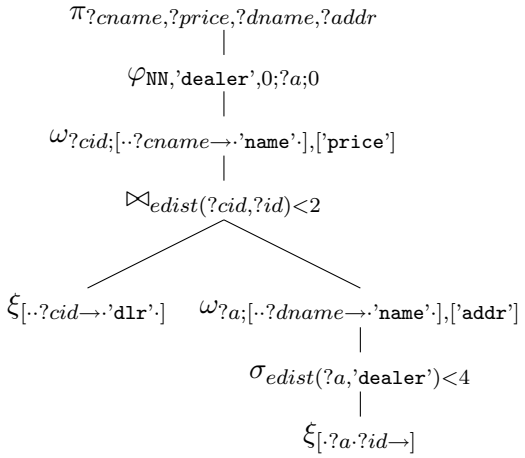
Now, assume there are some more relations about dealers. For instance, another participant might provide a relation such as:

- `stores`: (`deal,name,addr`) with `cars.dlr` → `deal`

VQL allows to query for all attribute names in an edit distance of 3 or below from `dealer`, which helps to handle typos or abbreviations. This reflects to a nearest neighbor query. The information found about dealers are joined with car triples by similarity on the ids. This lists all cars together with corresponding dealers. The final result is sorted according to the distance from `dealer`:

```
select ?cname,?price,?dname,?addr
where { ?d ?a ?id . ?d 'name' ?dname . ?d 'addr' ?addr .
        ?o 'dlr' ?cid . ?o 'name' ?cname . ?o 'price' ?price
        filter edist(?id,?cid) < 2
        filter edist(?a,'dealer') < 4}
order by ?a NN 'dealer'
```

This shows that queries can be formulated with limited knowledge about schemata – without having to explicitly query this knowledge before. A corresponding logical query plan is:



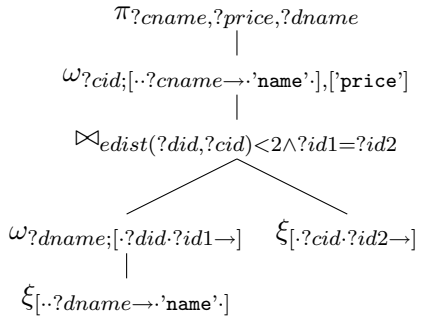
Note that the right-side similarity extraction will extract also the attribute `dlr`, because the distance to `dealer` is 3. But, these triples are also extracted on the left side. In this example, a self-join between triple sets containing attribute `dlr` is prevented by the materialize operation in the right branch of the join. All these triple sets are filtered out, because attribute `addr`

cannot be materialized for them. Of course, this is not always the case. We get back to this issue in the next example.

It is straightforward to formulate similarity joins on schema level. To select all cars with price and dealer names, whereby allowing an edit distance of 1 between the names of the join attributes, we can issue:

```
select ?cname,?price,?dname
where { ?d ?did ?id1 . ?d 'name' ?dname .
        ?o 'name' ?cname . ?o 'price' ?price . ?o ?cid ?id2
        filter edist(?cid,?did) < 2 filter ?id1 = ?id2}
```

This results in an algebra representation like:



This illustrates how the originally unbound triple pattern `?d ?did ?id1` is bound by another pattern that contains `?d`: the pattern `?d name ?dname`. Further, `id2` from the right branch of the join is bound by `id1` from the left side. Without these bindings, the query should be rejected. The example is well suited to illustrate the algebra's triple model again. Further, we can highlight several issues concerning attribute naming, self-joins, and optimization of query plans. Imagine the following example tuples and their triple representations:

cars	**name**	**hp**	**price**	**mil**	**dlr**
x	bmw	100	15000	5000	d1
y	bmw	90	20000	2000	d2

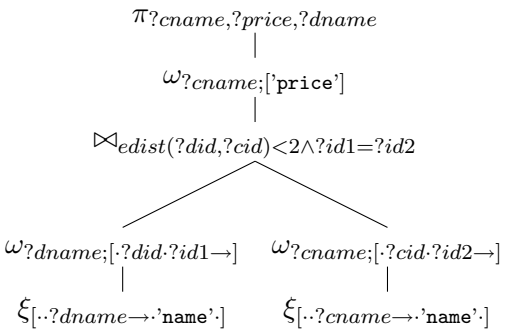
dealers	**id**	**name**	**addr**
v	d1	stephen	kc

stores	**deal**	**name**	**addr**
v	d2	jeremy	nj

⇒

(x name bmw) (x hp 100)
(x price 15000) (x mil 5000)
(x dlr d1)
(y name bmw) (y hp 90)
(y price 20000) (y mil 2000)
(y dlr d2)
(v id d1) (v name stephen)
(v addr kc)
(w deal d2) (w name jeremy)
(w addr nj)

An equivalent query plan to the one from above is:



If the query is evaluated using this query plan, both extract operations will produce the same triple sets. Only the stored variable references will differ in the variable names. We omit the variable references, because they are intuitive. The triple sets are:

$\{x$ name bmw$)\}$
$\{(y$ name bmw$)\}$
$\{(v$ name stephen$)\}$
$\{(w$ name jeremy$)\}$

The same holds for the input of the join. The resulting triple sets from both sides are:

$\{(x$ name bmw$), (x$ hp 100$), (x$ price 15000$), (x$ mil 5000$), (x$ dlr d1$)\}$
$\{(y$ name bmw$), (y$ hp 90$), (y$ price 20000$), (y$ mil 2000$), (y$ dlr d2$)\}$
$\{(v$ name stephen$), (v$ id d1$), (v$ addr kc$)\}$
$\{(w$ name jeremy$), (w$ deal d2$), (w$ addr nj$)\}$

Thus, the join operation will join also identical triples from both sides – like a self-join. The result of the join is the following:

$\{(x$ name bmw$), (x$ hp 100$), (x$ price 15000$), (x$ mil 5000$), (x$ dlr d1$)\}$
$\{(x$ name bmw$), (x$ hp 100$), (x$ price 15000$), (x$ mil 5000$), (x$ dlr d1$),$ $(v$ name stephen$), (v$ id d1$), (v$ addr kc$)\}$
$\{(y$ name bmw$), (y$ hp 90$), (y$ price 20000$), (y$ mil 2000$), (y$ dlr d2$)\}$
$\{(y$ name bmw$), (y$ hp 90$), (y$ price 20000$), (y$ mil 2000$), (y$ dlr d2$),$ $(w$ name jeremy$), (w$ deal d2$), (w$ addr nj$)\}$

Actually, the subsequent materialize operation is needless, because attribute `price` is already materialized. Finally, the projection returns the result of the query:

?cname	**?price**	**?dname**
bmw	15000	bmw
bmw	15000	stephen
bmw	20000	bmw
bmw	20000	jeremy

The result rows corresponding to the self-joined triple sets are obviously "wrong". This is due to the generic data model and the defined operator semantics. One way to filter or prevent this is to add appropriate methods in the application layer. Another way is to change the operator semantics. Besides this, the example highlights several other relevant issues:

1. It is a fortunate coincidence that the triples with attribute `id` are also accounted – the edit distance of 3 is only due to the fact that `id` and `dlr` are both rather short. This gives some hints concerning the appropriateness of the edit distance and raises the question if other similarity measures could fit better.

2. The example points out the importance of an adequate logical optimization process. For instance, if the final ω operator for attribute price is moved below the join operator, the self-join will be prevented. Another question is whether to use the ξ operator for binding or not.

3. Once again, the benefit of prefixes in attribute names is emphasized.

4. Besides the logical optimization, the example highlights several important issues we have to take into account when we discuss physical operators and the cost-based optimization process: In which way can and should we utilize the binding of variables? Can we identify possible groupings of operators in order to prevent extracting all triples stored in the system? How to identify and handle shared expressions and shared intermediate results? These and other questions are considered in Chapter 6.

5. Finally, the algebra and its semantics could be modified. So, a special join operator preventing self-joins (easy to detect based on the OIDs) could solve some issues from above.

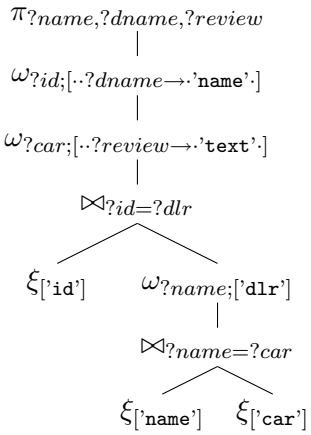
Last but not least, we introduce the formulation of a multiway join. Consider a third relation of car reviews:

- `review`: (`car`,`text`) with `car` → `cars.name`

The following query selects all cars with corresponding dealers and reviews:

```
select ?name,?dname,?review
where { ?x 'name' ?name . ?x 'dlr' ?id .
        ?y 'id' ?id . ?y 'name' ?dname .
        ?z 'car' ?name . ?z 'text' ?review }
```

A query plan containing this multiway join looks like:



These examples should be sufficient to clarify VQL and the underlying algebra. For completeness, we provide a last example including the skyline operator. This is also intended to motivate its relevance and usefulness:

```
select ?name
where { ?o 'hp' ?hp . ?o 'price' ?price . ?o 'mil' ?mil .
        ?o 'name' ?name}
order by skyline of ?hp MAX, ?price MIN, ?mil MIN
```

This intuitive example query searches for cars that are preferably powerful (`?hp`), cheap (`?price`), and have low mileage (`?mil`). The result represents the names of the cars that represent the best possible trade-off between all three ranking goals. A possible algebra representation is:

$$\pi_{?name}$$
$$|$$
$$\Phi_{\texttt{MAX,MIN,MIN};?hp,?price,?mil}$$
$$|$$
$$\omega_{?o;[\texttt{'price'}],[\texttt{'mil'}],[\texttt{'name'}]}$$
$$|$$
$$\xi_{[?o\cdot\cdot?hp\rightarrow\cdot\texttt{'hp'}\cdot]}$$

The algebra and its semantics proposed here (which already ran through several revisions) may need some more work for reaching standards like SQL and SPARQL. However, they are sufficient for the following discussions. Before we get over to the discussion of physical operators and query processing strategies, we present options for integrating semantic correspondences into query formulation.

5.1.1 Using Views for Representing Mappings

We discussed the representation of schema correspondences in Section 4.2.3. In this context, we mentioned also the option to model correspondences using a view-based approach. Now, after introducing the fundamentals of query language and algebra, we can get back to this point.

The view-based approach is known from several PDMS proposals using XML and other languages [84], but also from views in SQL. Using VQL, we could follow a similar approach. Assume we want to represent the following mappings:

- `buch⇐book`:
  - `titel⇐title`
  - `autor⇐author`
  - `preis⇐dollar`/(*result of query for EUR to USD rate*)
- `personen⇐person`:
  - `name⇐firstname||lastname`
  - `geb⇐`*today()*`-age` ∀ `age` ≥ `18`

Corresponding views could be formulated like:

```
create view buch<=book as
select ?title as titel, ?author as autor, ?dollar/?rate as preis
where { ?o 'title' ?title . ?o 'author' ?author . ?o 'dollar' ?dollar .
        ?p 'rate' ?rate . ?p 'source' 'EUR' . ?p 'target' 'USD'}
```

```
create view personen<=person as
select ?firstname||' '||?lastname as name, today()-?age as geb
where { ?o 'firstname' ?firstname . ?o 'lastname' ?lastname .
        ?o 'age' ?age filter ?age>=18 }
```

An advantage of this is that it is possible to represent mappings that make use of arbitrary functions, e.g., the concatenation of strings. When taking a closer look on this approach, several drawbacks get evident. So, the representation of arbitrary view definitions in a set of triples is challenging.

One way to model this view information in the triple model is to generate three triples, one for each part of the view definition. The following shows a possible mapping for attribute concatenation and the resulting triples:

```
create view personen:name as
select ?firstname||' '||?lastname as name
where { ?o 'firstname' ?firstname . ?o 'lastname' ?lastname}

(o mapping personen:name)
(o projection 'firstname|| ||lastname as name')
(o condition '?o firstname ?firstname . ?o lastname ?lastname')
```

Obviously, this uses the distributed index as a "dumb" string storage and involves repeated query parsing at query time to resolve these mappings. This implies unwanted costs. Moreover, expecting high-level query parsing and processing capabilities (i.e., arbitrary functions, such as concatenation in the example) at *each* peer is contrary to some essential ideas of this work, e.g., the light-weight character.

In the following, we focus on correspondences without complex functions. We restrict ourselves to basic correspondences as introduced in Section 4.2.3. We discuss a solution for direct mappings (like attribute A equals attribute B) and the support of ontology-based correspondences.

5.1.2 Integrating Semantic Correspondences

This section clarifies the basic approach for integrating schema correspondences into query formulation. Besides providing methods for tightly integrating the handling of heterogeneities, this extends the possibilities to formulate queries with limited knowledge about schemata. A logical query plan operator for integrating semantic correspondences into the flow of query processing is presented in the end of this section.

We use a simple example about services sharing personal and geographical data. We do not list the relations explicitly. It is sufficient to focus on the mappings between relations. Assume the following basic correspondences modeled as introduce before:

```
(person:name map:equiv people:name)
(person:hometown map:equiv people:livesin)
(de:city map:equiv gb:town)
(gb:town map:equiv fr:ville)
```

Imagine the following example query looking for all persons from German cities:

```
select ?name
where { ?o 'person:name' ?name . ?o 'person:hometown' ?city .
        ?o2 'de:city' ?city }
```

This assumes that relation `de` only stores German cities, which is a kind of implicit semantic knowledge. This fact does not hamper the illustration of the general idea, rather it helps to understand specific problems. Moreover, it shows that also such implicit semantics are supported.

Now, the correspondence information stored in the system can be used to rewrite the query in order to expand the result set accordingly. Usually, the question which correspondences when to apply is evaluated on a complex set of rules using inference techniques. In a distributed and decentralized environment this can be very expensive. The focus of this work is on the actual (and light-weight) integration of mapping information into distributed query processing. Thus, we use the introduced mappings as a special kind of inferences in order to implement a basic rewriting. Knowledge about several basic rules mandatory for this task is assumed. Details about special challenges on the logical level, such as resolving transitivity and reflexivity, different kinds and representations of correspondence information, and algorithms for finding correspondences are subject of other works. Insights gives, for instance, the GridVine system [6, 58]. An expanded query respecting the mappings from above will have a similar shape like this:

```
select ?name
where { ?o 'person:name' ?name . ?o 'person:hometown' ?city .
        ?o2 'de:city' ?city
    union
        ?o 'people:name' ?name . ?o 'people:livesin' ?city .
        ?o2 'de:city' ?city }
```

This assumes that also the `people` relation stores information about German citizens. The following query includes a third part. With this part we also query for transitive mappings. This exploits semantic knowledge even if actual correspondences are unknown at query formulation time. This can be seen as a sort of pay-as-you-go querying:

```
select ?name
where { ?o 'person:name' ?name . ?o 'person:hometown' ?city .
        ?o2 'de:city' ?city
    union
        ?o 'people:name' ?name . ?o 'people:livesin' ?city .
        ?o2 'de:city' ?city
    union
        'people:name' 'map:equiv' ?a1 . ?o ?a1 ?name .
        'people:livesin' 'map:equiv' ?a2 .
        ?o ?a2 ?city . ?o2 'de:city' ?city }
```

This would, for instance, include mappings like `(people:name map:equiv foaf:friend)` and `(people:livesin map:equiv foaf:place)` into query expansion at query runtime. Integrating additional ontology information is straightforward. Assume `person` belongs to a concept called `human` and we want to query also subconcepts, e.g., `celebrities`, materialized in a relation `whoiswho`.

The ontology information could be represented as follows:

```
(person ont:belongs human)
(person:name ont:attr:id human)
(person:hometown ont:attr:city human)
(celebrities ont:subclass human)
(whoiswho ont:belongs celebrities)
(whoiswho:givenName ont:attr:id celebrities)
(whoiswho:city ont:attr:city celebrities)
```

A query respecting this would look like:

```
select ?name
where { ?o 'person:name' ?name . ?o 'person:hometown' ?city .
        ?o2 'de:city' ?city
    union
        'person' 'ont:belongs' ?p . ?c 'ont:subclass' ?p .
        'person:name' ?attr1 ?p . ?a1 ?attr1 ?c .
        'person:hometown' ?attr2 ?p . ?a2 ?attr2 ?c .
        ?o ?a1 ?name . ?o ?a2 ?city . ?o2 'de:city' ?city
```

Like this, semantic expansion is integrated directly into query formulation and, as we show later, into query processing. But, this requires explicit semantic and expert knowledge in order to build these queries. This information must be available locally. We refer to this as *direct expansion*.

For a standard user, issuing queries from the user level (i.e., without the need for explicit knowledge about semantic information, its structure, and the actual integration into querying) is much more preferable. For this, we propose to use a kind of semi-automatic query rewriting. The query only specifies whether query expansion shall be triggered or not, optionally on which attributes. This can be extended by additional information, such as a maximal level of transitivity or an instruction to involve ontology data. We distinguish automatic and semi-automatic expansion. Both approaches extend the support for handling heterogeneities and limited knowledge even more. Further, the pay-as-you-go and ad-hoc characteristics are intensified.

Automatic Expansion The idea is to let the user or the application indicate whether semantic expansion should be triggered or not. Using automatic expansion this is done in an all-or-nothing manner. This means that either all known correspondences relevant to the query are integrated into query processing or none at all. This may be accompanied by a maximal level of transitivity or a direction for navigating in an ontology. We support this triggering with a notation similar to the prefix declaration in SPARQL:

```
expand
select ?name
where { ... }
```

This would be fine in most situations. But, this can be problematic if many mappings exist and if several attributes contained in the query should not be expanded. In the example from above, the expanded query would look like:

```
select ?name
where { ?o 'person:name' ?name . ?o 'person:hometown' ?city .
        ?o2 'de:city' ?city
    union
        ?o 'address:name' ?name . ?o 'address:livesin' ?city .
        ?o2 'de:city' ?city
    union
        ?o 'person:name' ?name . ?o 'person:hometown' ?city .
        ?o2 'gb:towns' ?city
    union
        ?o 'person:name' ?name . ?o 'person:hometown' ?city .
        ?o2 'fr:ville' ?city
    union
        ... }
```

The automatic expansion of `de:city` (including transitivity) results in persons from cities outside Germany (relations `gb` and `fr`) contained in the result as well, which may not be expected. This simple example reveals the possible desire for having more control on the degree of expansion.

Semi-Automatic Expansion An intuitive alternative is to directly indicate all attributes that should be mapped:

```
expand    person:name, 1
expand    person:hometown, 1
ontology  sub

select ?name
where { ... }
```

Or, with the support of wildcards:

```
expand    person:* 1
ontology  sub
select ?name
where { ... }
```

In the example, we also suggest a way to indicate the relations of an ontology that shall be regarded, here by providing `sub` to indicate the subclass relationship. There are two main problems with this approach: *(i)* correspondences for each attribute must exist in order to achieve meaningful results, and *(ii) each* of these attributes must be listed *explicitly* in the query as well. Nevertheless, we presently use this option, because it promises good flexibility and usability in parallel.

Further explanations on query rewriting, transitivity and complexity issues, views, etc. can be found in [138].

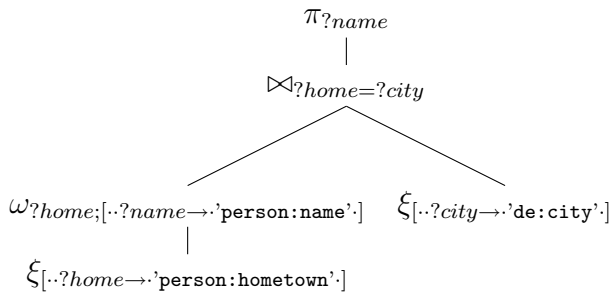


Figure 5.2: Unexpanded query

The Map Operator

Since we store semantic correspondences in the same way as we store raw data, mapping information can be queried using separate queries. After collecting all needed information, the original query can be expanded at the initiating peer. But, this is contrary to the aspired tight integration into stateless query processing. We suggest to achieve semantic expansion by integrating according subqueries into the original query plans. Like this, they are tightly integrated into the process of query planning and processing. (Semi-)Automatic query expansion provides a way of integrating correspondences without requiring sophisticated experience and knowledge. In query plans, this needs to be represented by an according operator. In this section, we introduce the map operator μ_{VTP}. In a query plan, this operator indicates the VTP for which semantic correspondences shall be resolved.

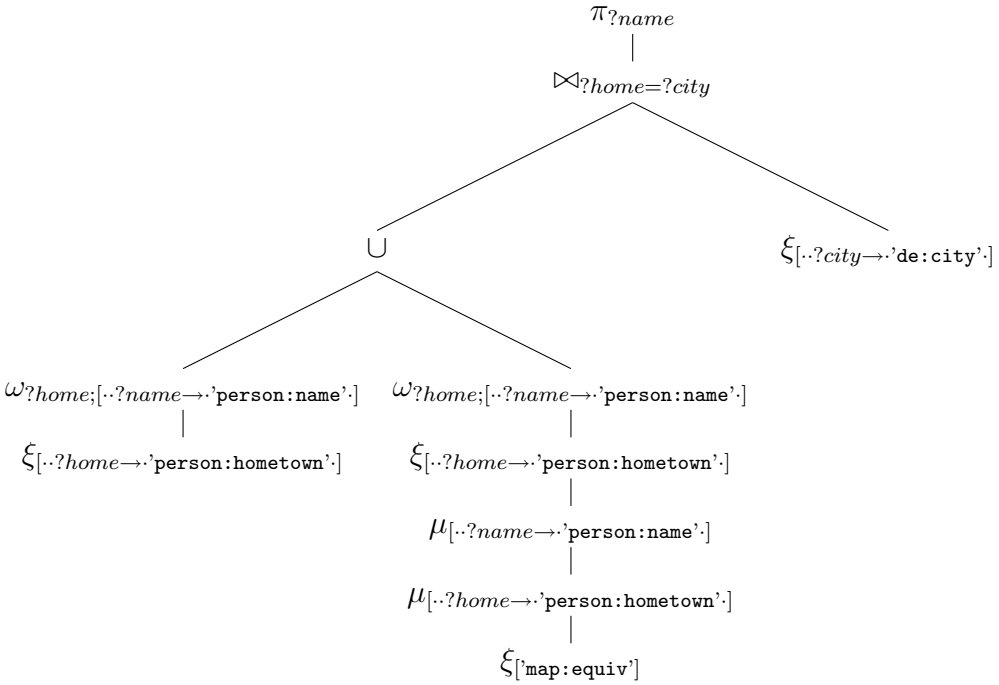


Figure 5.3: Query with mappings for expansion

Figure 5.2 shows the query plan corresponding to an unexpanded user query introduced before. If mapping information is directly integrated into a VQL query using according triple patterns,

these query parts result in separate subtrees of the corresponding query plan. We propose to enable semi-automatic expansion following the idea of adaptive query processing. In this case, the user is not required to formulate complex queries on mappings and ontologies. But, it results in analog subqueries as if it would be directly formulated.

Routing operators that refer to attributes to be mapped are copied to new subtrees of the query and accompanied by corresponding μ operators, as shown in Figure 5.3. Below these operators, subquery plans are inserted in order to materialize the information needed to resolve the mappings. In the case of simple equivalences, this may be just one operator extracting all triples on the `map:equiv` attribute. This is shown in Figure 5.3. As soon as the needed information is found, semantic expansion is finalized by modifying the appropriate operators in the same subtree (like ξ and ω in the example). The included VTPs are modified using the identified mapped attributes. The usage of a renaming operator is not needed. We just have to use the same variables. Like this, the map operator can be placed independently from other operators, as long as mappings are resolved below operators that use the according variables.

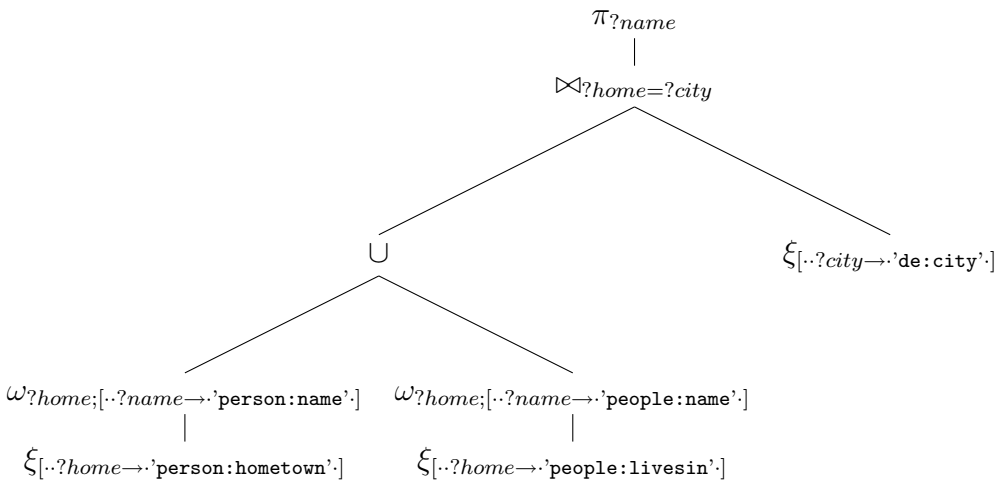


Figure 5.4: Expanded query

The resulting expanded query plan is shown in Figure 5.4. All operators above the union operator can handle triples belonging to `people` in the same way as they handle triples belonging to `person`.

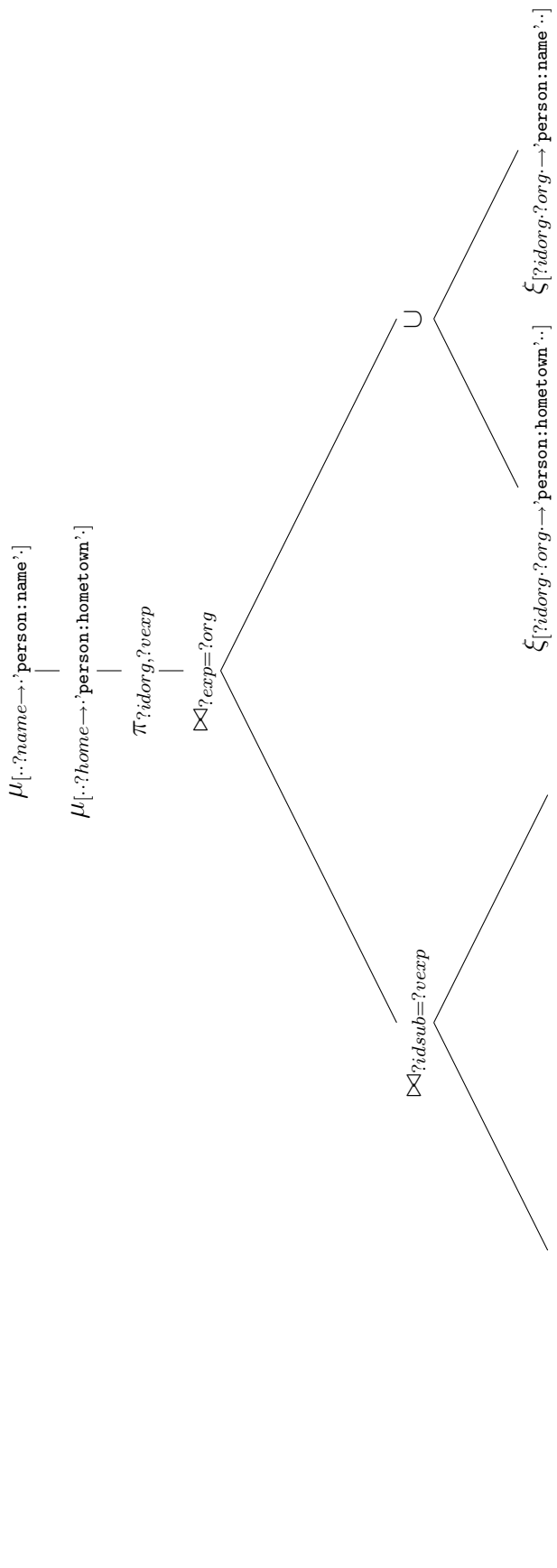


Figure 5.5: Expansion using ontology

As an extension, we can also query complex ontologies. This reflects to replacing the extraction for attribute `map:equiv` in Figure 5.3 by a more complex subquery plan, as shown in Figure 5.5. For readability, we only show the part of the query plan reflecting the mapping process. This would extract subclass information from an ontology-like data as:

```
(person ont:belongs people)
(person:name ont:attr:id people)
(person:hometown ont:attr:city people)
(celebrities ont:subclass people)
(whoiswho ont:belongs celebrities)
(whoiswho:givenName ont:attr:id celebrities)
(whoiswho:city ont:attr:city celebrities)
```

First, information on the ontology attributes `belongs` and `subclass` are extracted and joined accordingly. Afterwards, all attributes belonging to ontology concepts are extracted, using a prefix query on the commonly used prefix `ont:`. This could be extended using multiple different prefixes or similarity queries on schema level. Further, re-usage of shared expressions can help to optimize query processing. A join combines the so extracted attribute candidates and the concepts found before. Here, the usage of a prefix query avoids the problem of the unbound triple pattern `?o ?a ?v`. An alternative is to use a nested loop approach in the join operator (the left side is already materialized) or to use adaptive operators. Finally, to materialize all information needed for the actual mapping, a last join on schema level combines all candidates with all ontology triples for the attributes that shall be mapped (extracted using a lookup on corresponding OIDs). After that, the mapping operators can be processed accordingly. Note that this is only one of various possible query plans. It illustrates how the graph-based query evaluation approach known from RDF data is modeled using the database-like operators of this work.

5.2 Related Work

SPARQL [165] is a declarative query language for RDF that fulfills most of the requirements on a query language appropriate for the proposed infrastructure. SPARQL queries are evaluated following the triple model of RDF. It supports to define triple patterns, which specify the triples contained in the result. There is no mandatory `FROM` clause, as known from SQL, so this fits well to the idea of a universal relation. SPARQL supports variables and constants in triple patterns that are combined by conjunction. There may be other RDF languages fitting our needs as well. Our choice is on SPARQL, because it is a standard in the Semantic Web community. As we rely on a triple data model as well, SPARQL is well suited for our approach. We adopt a subset of all SPARQL functionalities. Features specifically designed for graph-based RDF query evaluation and Semantic Web applications are renounced. On the other hand, we extend the chosen subset by some functionality tailor-made for handling large-scale and very heterogeneous data collections, such as similarity queries, ranking queries, schema correspondences, etc. [140] gives an overview of similar RDF query languages.

For fulfilling the remaining requirements, and for reasons of convenience and practice, we base all extensions to SPARQL on the ideas of SQL [47]. This is intuitive, because SPARQL is already inspired by SQL, which in turn is the de facto standard language for relational databases. Although there are several discussions running on how exactly it conforms to the requirements of the relational algebra [81], probably each person dealing with databases in any

manner knows (parts) of SQL. Further, it is a well explored and continuously improved[1] query language reflecting the needs of general-purpose database applications. The same holds for the relational algebra [54], which is also used by all relational DBMS. The majority of all operators introduced below conforms to its definition in the relational algebra. These are the reasons why we use it as a guideline for the vertical query algebra.

There are several proposals discussing the integration of semantic correspondences into query processing. Those dealing with the actual processing techniques are introduced in Section 6.6. Alternatives for integrating semantics into query formulation are intensively discussed by the Semantic Web community. [116], for instance, describes how to extend SPARQL by semantic path expressions. Other works propose SPARQL extensions similar to the ones presented in this work, e.g., for ranking queries [192].

In Section 5.1.1 we briefly discussed how to utilize views to model semantic relations. A similar approach of using RDF-based views for representing semantic correspondences is presented in [117]. SQPeer uses RVL views [134] and processes RQL [110] queries on top of it. But, the authors assume globally agreed schemata. All sources that can map to a common schema are managed in one designated semantic overlay network. Views are used to implement a local-as-view approach.

XQuery [30] and XPath [52] are the standard query languages for XML data. [84] uses an XQuery-based language for defining mappings. The authors also discuss the definition of mappings between XML and RDF data. Consequently, XQuery is also used for formulating user queries. Basic XQuery and XPath queries can be mapped to VQL, as it is presented here. The triple model supports also mightier language constructs that would allow to provide full XQuery and XPath functionality. Actual efficiency of query processing and query expressiveness depend on the modeling of XML data in the triple store. [118] discusses related issues of P2P XML data management in general. An approach for indexing XPath expressions in DHT systems is proposed by [195]. Data model and infrastructure proposed in this work are ready-made for XML data management and querying, but require more specialized query and index constructs to support them in a meaningful way.

Pig Latin [154] is a language that shall bridge the gap between the declarative query language SQL and the procedural MapReduce [62] approach. Thus, it is rather designed for data processing than for data querying. It follows a parallel programming approach and is rather comparable to the relational algebra than to SQL. Pig Latin is designed for people with programming background to process computationally expensive tasks while staying to a familiar program logic. It is not as suited for people with database background, which are used to a declarative calculus character, as SQL and SPARQL are. However, it is closely related to the presented vertical algebra and cognitions and techniques can be useful to optimize and process query plans.

[1] http://www.jcc.com/sql.htm

Chapter 6

Physical Query Operators and Processing Strategies

Triple and index model provide a basis for discussing query processing. VQL, vertical algebra, and according mapping rules act as an interface to the physical query algebra. This chapter discusses the operators of the physical algebra and their mappings to logical operators. We present algorithms for operator implementations and discuss different query processing strategies. We further overview the process of physical optimization and provide fundamentals like cost estimations.

VQL queries are transformed into logical query plans as introduced in Chapter 5. These plans consist of only logical operators. To actually process the plans, logical operators have to be replaced by physical ones. This is matter of the query optimization. Usually, there exist different implementations for one logical operator. They differ in the degree of parallelism, the used indexes, and other significant performance factors. All implementations can use only the functionality provided by the underlying DHT. This comprises the basic put/get operations and, in cases, extended features like range and prefix queries.

This chapter first presents the different physical operators in Sections 6.1, 6.2, and 6.3. These operator implementations are main ingredients of the proposed framework. Afterwards, we discuss different strategies for processing complex query plans in Section 6.4. This involves an overview of query optimization and adaptive query processing. Finally, in Section 6.5, we provide cost estimations for the presented operators suitable for implementing a cost-based query optimization. Related work on query processing in large-scale P2P information systems is summarized in Section 6.6.

This chapter introduces and discusses valuable techniques to handle a couple of the mentioned challenges, such as:

- supporting scalability in data, queries, and participants by introducing query processing techniques specifically designed for widely distributed environments
- providing robustness and availability through parallel and stateless query processing
- achieving fairness and efficiency by distributing query load
- enabling efficient, ad-hoc, and in-situ querying with guarantees and predictable costs
- providing a basis for handling heterogeneity in data and query processing capabilities

Strategies for processing query plans are discussed in Section 6.4. Nevertheless, a general understanding is required to follow the explanations on operator implementations. We propose

to follow a strict query shipping approach [120]. Query plans are processed by shipping them to those peers responsible for a part of the data to process. This is done operator-wise, traversing query plans from the bottom to the top. But, instead of sending single operators, complete query plans travel the network. This enables a totally stateless fire-and-forget strategy, because all information about the state of a query are included in each query plan. Each peer processes as much operators of a query plan as it (currently) can. Whether an operator can be processed depends on the kind of operator, the amount of available input data, and the applied plan processing strategy. The peer generates and inserts according output data for each executed operator. As soon as the next operator in the chain of processing cannot be executed locally, the according query plan – containing operators to process and result data of operators already processed – is forwarded to the next peer(s). Thus, the DHT serves as a hash table for storing and indexing data and as a content-addressable network for routing query plans. To enable parallelism, query plans can be cloned and shipped independently. In large-scale distributed data systems, the main optimization objectives that should be regarded for query processing are in the given order:

1. Minimize the number of generated messages.
2. Decrease query answer times and increase robustness.
3. Reduce bandwidth consumption.
4. Optimize local processing costs.

We mainly concern objectives 1 and 2. Answer times are very hard to predict and are covered by regarding the number of query hops. Objective 3 usually comes along with 1. We indicate the points where this needs special concern. Objective 4 is only secondary and a matter of general algorithmic optimizations, which are discussed thoroughly in other works.

We already proposed a couple of special query implementations in prior works [106, 107]. They are also based on the main principles and techniques proposed here, but do not adhere to the strict operator encapsulation. As a consequence, they imply inexpedient waiting states. Although this can bear advantages, such as fewer duplicates in the final result, we experienced that a strict stateless processing should be preferred.

6.1 Basic Query Operators

As outlined before, routing of a query plan depends on the query operator that shall be processed next. To achieve encapsulation and independence of operators, we propose to initiate according routing processes in the operators themselves. Corresponding operators are called *routing operators*. A routing operator determines the peer(s) to route to by hashing. For instance, if triples of a specific OID o are needed, the operator can route the query to the peer responsible for the hash key $h(o)$. Several operators, such as filtering selections, can be processed in a sequence without intermediate routing. This is provided if data needed for processing are already available at the current peer. Such operators are called *local operators*. An example is a selection σ following an extraction ξ, which in turn is a routing operator. After routing and extracting data, no further data are needed to process the selection. Thus, the corresponding operator can be processed locally.

Routing operators are restricted to use DHT functionality. If only a single hash key is looked up, this will be done using the get operation. However, in the usual case a single routing process will target multiple peers, e.g., if all triples belonging to a requested attribute are spread

over multiple peers. In this case, the DHT provides several alternatives for corresponding implementations. *Sequential routing* looks up all requested keys or key partitions in sequence and is applied by *sequential operators*. Looking up all keys or partitions in parallel promises to be more feasible in dynamic and unreliable environments. We call such operators *intra-operator parallel*, because parallelism is applied internally in one operator.

We follow a notation known from relational databases to distinguish different operator implementations. For referencing operators, we use Greek letters as introduced in Section 5.1. Physical implementations are indicated by a superscript as follows:

- op^{LOC}: local operator *op*
- op^{SEQ}: sequential operator *op*
- op^{PAR}: intra-operator parallel operator *op*

This general notation is extended below to refer to concrete operator implementations.

We introduce physical operators in the same order as we introduced the matching logical operators. The current section deals with standard operator functionality. This involves the operators adopted from the relational algebra ($\pi, \sigma, \times, \bowtie$) and those specifically designed to meet the triple model (ξ, ω). In the context of the mentioned optimization objectives, our main focus is on the crucial question: How to route query plans to peers responsible for processing parts of them? This mainly concerns the applied indexes and the degree of parallelism. Intuitively, $\pi, \sigma, \times$, and $\bowtie$ can usually be processed locally – responsibility for routing to the required data is due to extract and materialize operators beneath them. This holds as long as operators are not combined to achieve better performance (e.g., σ and ξ for similarity extraction). In Section 6.2, we extend the set of operators by such advanced operators that usually apply special indexes. Further, we introduce ranking operators in Section 6.2, which are based on more complex algorithms. In the current section, we assume availability of the three standard variants OID index, attribute index, and attribute-value index. We introduced these indexes in Section 4.2.1. We will note which operator relies on which index. If that index is not available, the specific operators are not available as well.

We begin with standard operators that are usually implemented as local operators.

π: The projection operator is always located on top of a query plan. Thus, it is the last operator that is processed before a final reply is sent to the query initiator. We suggest to always use it as the root of a plan, because it is the only operator that breaks the triple model. A set of variables – a result row – is projected from the input triple set. In the usual case, a projected result row is significantly smaller than the corresponding triple set from RT. The contrary can only occur if variables are projected several times. This is supported by VQL, but intuitively it is a very unlikely case. Thus, to save bandwidth, π should always be processed locally when all operators beneath it are done. Consequently, we support only one implementation: π^{LOC}.

The task of this operator is to resolve the variable references in order to project the correct values. As long as these references are correctly maintained during query processing, this task is straightforward. This is why we omit further details. How to maintain the references is a low-level implementation issue and described in Section 8.1. Note that a `distinct` projection in conjunction with parallel processing requires a final duplicate detection at the query initiator. Duplicates can be eliminated in each independent query plan copy, but access to *all* query plans is only guaranteed at the initiator.

σ: Selections are handled similar to projections. If σ is a stand-alone operator, all input data will be available locally. Thus, they are processed locally. Following, σ^{LOC} is the standard implementation. There are special implementations for combining σ with ξ and ω. These implementations are introduced in Section 6.2.

σ supports arbitrary selection predicates based on the standard operations for comparison $<, \leq, >, \geq, =$, and $\neq$. Further, some functions are supported, such as $edist()$ for evaluating similarity on string values. To keep query processing light-weight and independent from sophisticated capabilities, the list of supported functions should be kept small. However, if more complex functions and predicates are desired, they could be supported by a $\sigma^{SEQ/PAR}$ as well. The idea is to route the query to a peer capable of evaluating the predicate (e.g., the initiating peer where the predicate was formulated) for processing the selection. We restrict our approach to standard predicates that can be evaluated at each participating peer. Thus, as long as it is not combined with other operators, we restrict to σ^{LOC}.

$\times, \bowtie$: Similar circumstances as for π and σ hold for the join operators $\times$ and $\bowtie$. $\times$ joins triple sets without evaluating any condition. Thus, it is straightforward to implement it as a local operator $\times^{LOC}$ that is processed as soon as needed input data are available. Note that this does not require complete input data. The join operation can be processed also on parts of the input. The resulting sets of triple sets can be merged later on. We illustrate this on the basis of a small example when discussing query plan processing in Section 6.4. As all variants of $\bowtie$ were introduced as a combination of σ and $\times$, $\bowtie^{LOC}$ corresponds to a combination of σ^{LOC} and $\times^{LOC}$.

φ, Φ: Intuitively, ranking operators can be implemented as local operators φ^{LOC} and Φ^{LOC} straightforward. Routing operators on lower levels of the query plan are used to collect all needed data. But, if parallelism is applied on the lower levels, multiple copies of a query plan are processed concurrently at query time. Each of these copies will contain different data. In this case, we have to synchronize the plans before generating a final result. To achieve this, we have to define a synchronization peer before splitting a plan into different copies. Note that we have to do this using a unique identifier like an IP address. Providing a P-Grid path will not work, because several peers are responsible for the same path due to the applied replication. All peers that start processing of the ranking operator in any of the plan copies have to route their copies containing intermediate results to the same synchronization peer. Actually, the synchronization peer processes a kind of local ranking operator. But, to highlight the fact that input data are split over multiple incoming query plans, we write φ^{SYNC} and Φ^{SYNC}. These operators are finally processed on the synchronization peer and either *(i)* produce multiple result revisions (*online* processing) or *(ii)* block until a complete input is achieved (*blocking* processing). These two choices need some further investigation:

(i) With every incoming plan copy the set of intermediate results grows. The updated set can be ranked and optionally limited to N with each incoming copy. If a top-N query is processed, each peer sending an intermediate result should rank and limit the set of triple sets before sending. This can be further optimized by using a tree hierarchy. Such trees are a popular tool for aggregation queries [44, 99] and are not discussed here any further. At the synchronization peer, resulting query plans (containing only the ranked results) can be processed immediately or multiple intermediate results can be gathered first. In both cases, each time the final ranking is processed, a new *revision* is produced, which is recorded accordingly. For naive

online processing, a new revision can be generated each time a given time interval has passed. Another approach is to generate a new revision after a certain fraction of the input is received. This provides the transition from online processing to blocking processing, where the required completeness is set to 100%.

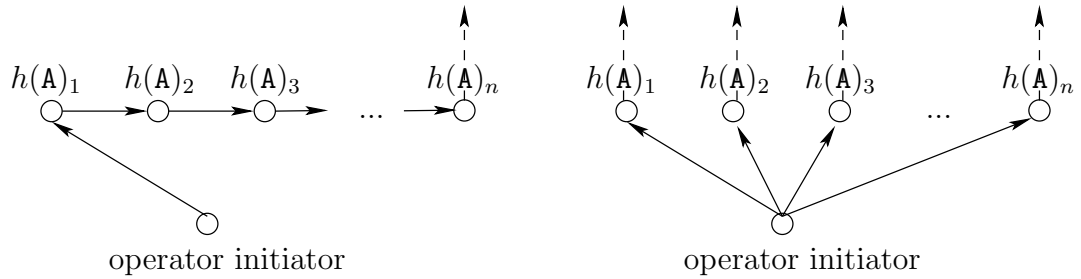
(ii) A blocking operator waits until a complete input is achieved. A main problem is that we generally do not know how many plan copies we have to expect. Thus, with every incoming plan we cannot decide about the current completeness of the intermediate result. We explain this further in Chapter 7, where we also provide an approximated solution for this. Without such a completeness estimation, blocking can only be implemented on the basis of constant time intervals. This can result in unnecessary long waiting times, but also in missed input data. The dynamic and unreliable behavior of participating peers may lead to large delays between consecutively arriving plan copies. This advises to generally prefer online processing over blocking processing. Of course, online processing on the basis of a required completeness below 100% suffers from the same problems and also requires mechanisms for estimating query completeness.

We introduce more implementations of ranking operators in Section 6.2.2. These variants try to minimize the effort for materializing the input for ranking by pruning the regarded search space accordingly.

So far, we introduced primarily local implementations for operators adopted from the relational algebra. These operators are easy to implement as local variants. Our focus is on the underlying routing operators, which are used to collect the data needed to process local operators. Further strategies for distributed processing of standard operators in P2P databases, especially for joins including rehashing and more, are focus of other works [99]. They adapt strategies known from centralized and distributed databases. Such techniques can be used to also optimize local query processing at each peer. For example, we do not take a merge join for distributed join processing into account. Even if sorted access would be available, this technique would involve too much waiting states and bidirectional message passing for synchronization. But, the method can be very useful to locally process (parts of) joins efficiently.

The routing operators ξ and ω are very similar. While ξ extracts triples conforming to a VTP and generates triple sets at the leaf level of query plans, ω materializes additional triples for existing triple sets at higher levels. Consequently, also implementation variants for both operators are similar. We first explain algorithmic details of how to handle the most common VTP ['A'] for extracting or materializing triples belonging to an attribute A. Afterwards, we generalize these algorithms for handling arbitrary VTPs. There is no local variant for both operators, because they always require to route query plans to responsible peers. Of course, a peer that starts the processing of such an operator may also be responsible for (parts of) the required data. In this special case we say the peer routes a query plan to itself. That this involves no extra messages is provided by the DHT. Like this, we can adhere to the same principle of operator processing as for the usual case.

ξ: For extracting all triples that belong to an attribute A, we can to route the query to all peers responsible for $h(\texttt{A})$. One approach is to route the query to all these peers in sequence. The sequential operator implementing this is called ξ^{SEQ}. Figure 6.1(a) illustrates the principle of this implementation. The peer that starts the processing of the operator routes to the first peer responsible for $h(\texttt{A})$. This peer can be identified using DHT functionality provided by `forward(first-key(A))`. At that peer, the operator is processed and afterwards the query is forwarded to the next peer in the sequence. This



(a) Sequential execution ξ^{SEQ} (b) Parallel execution ξ^{RQ}

Figure 6.1: Routing schemes for $\xi_{['A']}$

is repeated at all involved peers. At the end of this sequence (which can be determined similar to its beginning), we can finish the processing of ξ and continue with the next query operator.

Algorithm 1: General processing of $\xi^{SEQ}_{['A']}$

```
   Input: attribute name A
 1 if Π_local ⊆ h(A) ∨ Π_local ⊃ h(A) then
 2 |   forall t ∈ OT_local(t[a] = A) do
 3 |   |   add-to-result({t});
 4 |   end
 5 |   if Π_local ⊆ last-key(h(A)) then
 6 |   |   signalize(DONE);
 7 |   end
 8 |   else
 9 |   |   forward(next-key(Π_local, h(A)));
10 |   end
11 end
12 else
13 |   forward(first-key(h(A)));
14 end
```

The parallel alternative is illustrated in Figure 6.1(b). Here, each peer responsible for a part of $h(\mathtt{A})$ is contacted in parallel. After processing ξ, each involved peer signalizes that the operator is finished and continues with the next operator. This results in multiple plan copies, each containing different parts of the intermediate result. These plans are processed independently and finally merged at the query initiator. This works straightforward as long as the query plans do not contain operators that require synchronization, such as a ranking operator. For routing one plan copy to each responsible peer we can utilize DHT functionality. $h(\mathtt{A})$ is a prefix of the hash keys $h(\mathtt{A}||v)$ used to build the attribute-value index. If prefix queries are supported, we will use them. If they are not available, we map the prefix to an according range and issue a range-query routing. For mapping a prefix to a range, we can use the methods `first-key(A)` and `last-key(A)` provided by the DHT. Throughout this work, we write op^{RQ} for each operator op that utilizes range query mechanisms in the same way as ξ^{RQ} does here. This is to distinguish them from other parallel operators introduced later on. Nevertheless, ξ^{RQ} is parallel – querying a range in sequence is implemented in operators like ξ^{SEQ}.

Algorithm 2: General processing of $\xi^{RQ}_{[\texttt{'A'}]}$

```
   Input: attribute name A
1  if Π_local ⊆ h(A) ∨ Π_local ⊃ h(A) then
2  |  forall t ∈ OT_local(t[a] = A) do
3  |  |  add-to-result({t});
4  |  end
5  |  signalize(DONE);
6  end
7  else
8  |  range-forward(h(A));
9  end
```

We present the general algorithms for ξ^{SEQ} and ξ^{RQ} in Algorithm 1, in Algorithm 2 respectively. These algorithms are executed at each peer involved in processing the current operator. Π denotes a P-Grid path assigned to a peer. Π_{local} is the path of the current peer, i.e., the prefix hash key that the current peer is responsible for. Similar, OT_{local} refers to all triples that are available locally at the current peer. $\subset$ refers to the prefix relationship between keys. A peer may be responsible for all triples belonging to an attribute `A`. In this case the peer's path is a prefix of or equal to $h(\texttt{A})$, i.e., $\Pi_{local} \subseteq h(\texttt{A})$. This means, the "sequence" consists of only one peer. On the contrary, if $h(\texttt{A})$ is a prefix of the peer's path, i.e., $\Pi_{local} \supset h(\texttt{A})$, the peer will be responsible for only a part of all triples belonging to `A`. According to the assumptions from Section 4.1, only attribute names in the attribute-value index can result in hash keys that are prefixes of peers' paths. Note that this implies that $\Pi_{local} \supset$ last-key(`A`) cannot occur.

Most of the used function calls refer to DHT functionality and were introduced in Section 4.1. For convenience, we list them again together with additional functions and their special meaning in the algorithms:

- `add-to-result(`$\{t\}$`)`: adds a triple set $\{t\}$ to the set of resulting triple sets
- `next-key(`$\Pi, h(\texttt{A})$`)`: determines the next key for attribute `A` with respect to the current key Π (easy to implement using knowledge about direct neighbors)
- `first-key(`$h(\texttt{A})$`)`: determines the first key for attribute `A`
- `last-key(`$h(\texttt{A})$`)`: determines the last key for attribute `A`
- `forward(`k`)`: forwards the whole query to the peer responsible for hash key k and starts processing of the current operator there, i.e., with the same parameters
- `range-forward(`$h(\texttt{A})$`)`: forwards the whole query to all peers responsible for hash key $h(\texttt{A})$ and starts processing of the current operator there (using prefix routing or range query routing, depending on availability)
- `signalize(`$DONE$`)`: finishes the processing of the current operator and signalizes that

Line 13, line 8 respectively, is only executed by the peer initiating the processing of the operator. All other peers will execute only lines 2 to 10, lines 2 to 5 respectively.

ω: Two implementations of ω, namely ω^{SEQ} and ω^{RQ}, work analog to the corresponding ξ operators illustrated in Figure 6.1 and Algorithms 1 and 2. The difference is that they

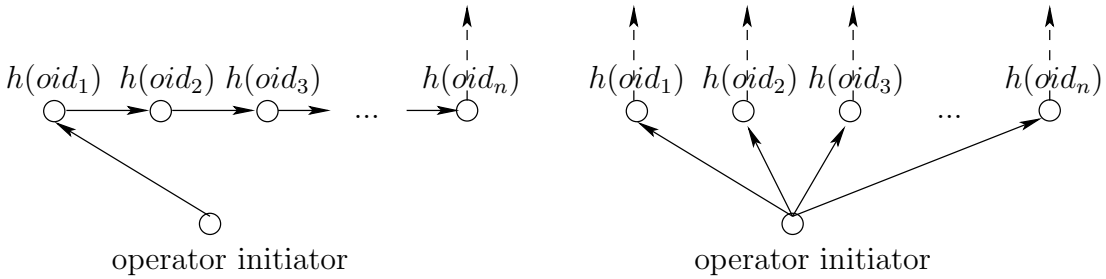
Algorithm 3: General processing of $\omega^{HASH,PAR}_{?r;['A']}$

```
   Input: reference variable ?r, attribute name A, input T[, OID o, triple set ts]
 1 if o = NULL then
 2 |   forall ts ∈ T do
 3 |   |   o = oid(?r, ts);
 4 |   |   forward(h(o), o, ts);
 5 |   end
 6 end
 7 else
 8 |   forall t ∈ OT_local(t[oid] = o ∧ t[a] = A) do
 9 |   |   add-to-result(ts ∪ t);
10 |   end
11 end
```

materialize only triples with matching OIDs and that they merge corresponding triples into triple sets before calling `add-to-result()`.

Due to the fact that ω works on a set of input triple sets T, we can implement a third physical variant ω^{HASH}. In contrast to ω^{SEQ} and ω^{RQ}, in which we route for attribute names, in ω^{HASH} we utilize the OID index. We extract the OID o from each input triple set (using the reference variable $?r$) and route to $h(o)$, where we can materialize all needed attributes. Although we natively base all routing processes on hashing, we call this version of the operator a *hash materialization* ω^{HASH}. An advantage of this is that we can materialize all attributes for one OID without having to route repeatedly. Using ω^{SEQ} or ω^{RQ} the query would be routed for all attributes in sequence, i.e., multiple operators would be inserted, each materializing one attribute. Note that materializing the attributes in sequence is mandatory. Otherwise, we would need extra synchronization in order to generate complete triple sets. Further, ω^{HASH} is the only possible implementation for the special operator $\omega_{?r;*}$. The fact that all triples with the same OID are stored at the same peer allows to efficiently materialize all existing attributes without having to know them a priori.

We illustrate ω^{HASH} in Figure 6.2 and present the underlying algorithm for $\omega^{HASH,PAR}_{?r;['A']}$ in Algorithm 3. In the algorithm, lines 2 to 5 are executed only by the initiator of the operator. The initiator determines the OID o for each input triple set ts. o and ts are inserted into plan copies before forwarding them, indicated by the `forward()` method with additional parameters. In the algorithm, we use o and ts to distinguish between the part that the initiator executes and the part that receiving peers execute. Thus, all peers responsible for requested OIDs execute only lines 8 to 10. Using $\omega^{HASH,PAR}$, all resulting query plans are sent in parallel. Alternatively, we could route to the peer responsible for the first OID, process the operator there, forward to the peer responsible for the next OID, etc. This corresponds to the sequential implementation $\omega^{HASH,SEQ}$. We omit that algorithm, because it differs only slightly and adheres to the processing principle illustrated in Algorithm 1. One obvious modification is that each peer that receives the query should process *all* OIDs it is responsible for, not only the one identified by the current value of o. This prevents from routing the query to the same peer repeatedly during the sequence. Using $\omega^{HASH,PAR}$ this can be exploited only in limits. Basically, we can combine routings for different OIDs that map to equal hash keys in one message, which is rather unlikely to occur. But, we can use more knowledge that is available at the peer that initiates routing. From information stored in the local routing table (common



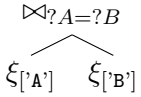
(a) Sequential execution $\omega^{HASH,SEQ}$ (b) Parallel execution $\omega^{HASH,PAR}$

Figure 6.2: Routing schemes for $\omega^{HASH}_{?r;[\texttt{'A'}]}$

for each DHT) we can identify the paths of known neighbor peers. If the path of a peer p is prefix of $h(o)$ for several different OIDs o, we can collect them in one message to p. This at least exploits the local knowledge to its full extent. In contrast, $\omega^{HASH,PAR}$ bears another advantage. Since we directly route for a specific input triple set, we have to include only this input and can discard all other data from the according plan copy. Compared to the other variants, which require each query plan to contain all input data, this reduces the bandwidth overhead

Note that we omit the maintenance of variable references in all algorithms. They are handled according to the explanations in Section 5.1 and are omitted for convenience. For supporting arbitrary VTPs $[?o{\cdot}?a{\cdot}?v \rightarrow \texttt{o}\cdot\texttt{a}\cdot\texttt{v}]$ we can apply the same principles. If o or v are given, ξ^{SEQ} will become a single key lookup. Using a range query as in ξ^{RQ} is not reasonable in this case. A range-query routing is a good choice only when we can expect multiple peers responsible for the queried key, for example, when materializing an attribute A using $h(\texttt{A})$. Further, ξ and ω can be combined with a subsequent σ_P. To exactly clarify which operators are combined and to conserve the structure of the query plan, we refer to such an implementation using both original logical operators. In the following, we use this two (or more) operator representation for all combined operator implementations. For instance, an operator combination like $\sigma_{x\leq ?A\leq y}(\xi_{[\texttt{'A'}]})$ should be processed by issuing an according range query from $h(\texttt{A}||x)$ to $h(\texttt{A}||y)$ rather than querying for $h(\texttt{A})$ and processing σ locally. We do not list all the possible combinations, because most of them are intuitive. With an according predicate P in σ_P, it may also be possible to extract continuous ranges from P. Then, we can use one ξ^{RQ} for each range, one ω^{RQ} respectively. Further, a loop approach as in ω^{HASH} can be used, for example, if a predicate like $?v \in \{v_1, v_2, \ldots, v_n\}$ is provided. This is possible also for the ξ operator. But, the parallel variants should be applied with care. If multiple triples are materialized for one triple set, these variants will require some extra synchronization in order to create complete triple sets. This is not needed for the sequential operators. This is related to the problem of materializing multiple attributes for one OID and the chance to save messages when processing sequences, both mentioned above.

Due to its binary character, $\bowtie$ can also be implemented as a routing operator, following the hashing principle introduced with ω^{HASH}. In the local variant $\bowtie^{LOC}$ we expect both of the input sides completely extracted and materialized. If one side of the join consists of only an extract operation ξ, it can be combined with the join. This works for equality predicates in the join. In this case, we can lookup the triples matching the join condition separately using the corresponding index. For instance, for a simple join like



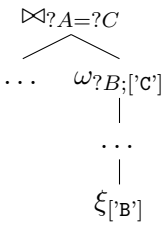
we can use the attribute-value index. Assuming that the left side is extracted first, all generated triple sets with attribute A can be used as input for the hash join. We can lookup $h(\texttt{B}||\texttt{a})$ for each value a of attribute A. Thus, we use hashing to find equal values. This corresponds to a hash join as known from relational databases. In contrast, we do not have to explicitly hash one of the input sides – this is already provided by the established indexes. If all relevant hash keys are queried in a sequence, we will write this operator $\bowtie^{HASH,SEQ}$. If all corresponding messages are sent in parallel, we will write $\bowtie^{HASH,PAR}$. We show the algorithm for $\bowtie^{HASH,PAR}_{?A=?B}(\xi_{['B']})$ in Algorithm 4. It follows the same principle as ω^{HASH}, but instead of looking up the OID of input triple sets we lookup attribute values. Lines 2 to 5 are executed only by the peer initiating the routing. In contrast, lines 8 to 10 are executed only by peers responsible for a part of $h(\texttt{B})$. In analogy to the OID o in Algorithm 3, we use the current attribute value v_A as an optional parameter for distinguishing between initiating and receiving peers.

Algorithm 4: General processing of $\bowtie^{HASH,PAR}_{?A=?B}(\xi_{['B']})$

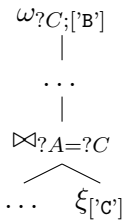
Input: variable name $?A$, attribute name B, left side input T[, attribute value v_A, triple set ts]

```
1  if v_A = NULL then
2  |   forall ts ∈ T do
3  |   |   v_A = ref(?A, ts);
4  |   |   forward(h(B||v_A), v_A, ts);
5  |   end
6  end
7  else
8  |   forall t ∈ OT_local(t[v] = v_A ∧ t[a] = B) do
9  |   |   add-to-result(ts ∪ {t});
10 |   end
11 end
```

On the first look, the concept of $\bowtie^{HASH}$ fits only for join operators with at least one ξ child. But, it is feasible also for a combination with the routing operator ω. Intuitively, a query plan like



would be rewritten to



in order to enable the usage of a hash join. But, especially in the context of more complex subqueries, it can improve performance if the whole subquery is kept below the join operator. Such a plan can still be processed using a hash join. The whole subquery except the ω operator has to be processed before starting the join operation. Then, ω is combined with $\bowtie^{HASH}$ the following: each input value `a` from the left side is used to route for $h(\texttt{C}||\texttt{a})$ as introduced above. At the corresponding peer(s), only values that match the join condition *and* an OID that is referred by $?B$ are joined. Following, $\bowtie^{HASH}$ can be chosen for arbitrary subtrees, as long as an ω operator is a direct child. This meaningfully extends the options for physical query planning. The advanced physical operators described next in Section 6.2 follow the same idea of combining a selective operator with an extraction ξ or a materialization ω.

The hashing concept provides, beside local, parallel range, and sequential range processing, a general implementation variant for physical operators. We discuss the costs of physical operators later on. But, some general rules are obvious. A hashing approach should be preferred only if one of the input sides is small. Otherwise, too much messages will be generated, which results in too much overhead. Parallel operators should be preferred over sequential operators in highly dynamic and unreliable environments. If a peer fails or a query message gets lost, this will not influence the progress of other plans processed in parallel. Sequential execution is vulnerable to slow or failing peers. This can result in delays or even query aborts due to broken chains of forwarding. But, these advantages of parallel operators are not for free. They come on the costs of more messages and thus, more load. This is particularly an issue if a query contains multiple parallel operators. The number of messages increases exponentially with the number of series connected parallel operators. We discuss this issue later on again. The exact differences between operator implementations are captured by the cost model that we introduce in Section 6.5. The estimated costs help to deduce which variant has the most benefits in which situation.

6.2 Advanced Query Operators

In this section, we present several physical operators that extend the set of basic query operators. These operators do not implement more functionality. Rather, they provide alternatives to the basic operators. They are designed to increase the efficiency of query processing by utilizing special indexes or sophisticated algorithms.

A crucial question in this context is: What to do if a specific index is not available? Moreover, how to detect which indexes exist? If an index is not built, a peer will recognize it when inserting and indexing local data. But, indexes can be restricted to specific attributes or values. Since all message routing is based on hashing, a lookup for a specific index entry will be routed to a valid source, all the same whether the dedicated (part of an) index exists or not. There is always a peer guaranteed to be responsible for the corresponding hash key. But, if the index does not exist, the expected triples will probably not be stored at that peer. Missing indexes can be handled only by some mechanisms like acknowledgment messages, which require knowledge about existing indexes at the contacted peers. Local index items and lookup messages can be

annotated accordingly. We do not discuss this issue further and assume all required indexes to be available on all values.

The presence of multiple indexes may cause duplicated results. This is because each peer can be responsible for parts of several indexes. Thus, on a peer we can find more than expected. For instance, if we use the value index to route to a peer, that peer may also hold relevant triples indexed by their OIDs. These triples are also available at some other peer that is responsible for another part of the value index. If all local data is used without filtering for the queried index, this can cause duplicates. The more indexes exist, the higher the rate of duplicates will be. Filtering is only possible if the index items are annotated accordingly, which causes small storage overhead. Duplicates increase robustness of query processing, because they represent a kind of replication on query level in addition to replication on peer and data level. There are two options: *(i)* include all local data at the risk of duplicated results, but at the gain of increased robustness. *(ii)* use only what was actually requested (i.e., by annotating index items) on the gain of less duplicates, but decreased robustness. The proposed operators are suited for both approaches. This should not be mistaken with duplicates that result from querying the same index items repeatedly in independent query plans. This can happen when parallel operators are used.

6.2.1 Similarity Operators

We motivated the indispensable need for similarity queries in the beginning of this work. Fuzzy queries based on similarity metrics will make the majority in Public Data Management applications. Unfortunately, they also represent a specifically expensive query type, because usually large amounts of data have to be considered. The physical operators introduced up to here are capable of handling similarity predicates. But, large amounts of triples have to be extracted and materialized before they can be filtered locally. This will involve a large fraction of all peers and implies an according large amount of messages. Usually, only a fraction of the queried peers will actually contribute to the final result. Similarity queries on instance level usually require to extract all triples belonging to a provided attribute. Similarity on schema level could only be applied – theoretically – when extracting all triples that exist in the system. Thus, these approaches can become rather expensive. In centralized database systems, special indexes are used to accelerate the processing of similarity queries and to reduce the resulting costs. We propose an analog approach. First, we briefly introduce the regarded similarity measures. Afterwards, we focus on evaluating string similarity efficiently. Finally, we suggest an according index and physical operators, suited for string and numerical data.

Similarity Measures

A typical distance measure for numerical data is the L_p or Minkowski distance. It is defined on two points $X(x_1, x_2, \ldots, x_n)$ and $Y(y_1, y_2, \ldots, y_n)$ from an n-dimensional space as follows:

$$L_p(X,Y) := \sqrt[p]{\sum_{i=1}^{n} (x_i - y_i)^p}$$

There exist special cases of this distance measure, such as the Manhattan distance for $p = 1$ and the Chebychew distance for $p = \infty$. We focus on the most popular Euclidean distance for $p = 2$. For one-dimensional points (e.g., values of a single attribute `A`), this becomes the simple difference $|X - Y|$. In this case, it can be mapped to range queries. Range queries have received increased attention recently and several DHTs can handle them already. We briefly discussed

this in Section 4.1. But, string data will be dominating in most of the aimed applications. This is specifically true for the schema level, but holds also for the instance level. For similarity measures on string values the situation is different. Usually, they cannot be mapped to range queries. This is especially true for the popular *Levenshtein distance* or edit distance [126]. In its simplest form, the edit distance of two strings s_1 and s_2 is the number of operations (insertion, deletion, or substitution of characters) needed to transform s_1 into s_2. For instance, the edit distance of s_1 = "edna" and s_2 = "eden" is 2:

- delete 'a'
- insert 'e'

Calculation of the distance can be achieved by applying a dynamic programming algorithm. In principle, any distance measure $d(x, y) \rightarrow \Re$ is supported by our approach. When routing operators ξ and ω are used to extract and materialize all corresponding candidate triples, any predicate can be processed in a local selection σ^{LOC}. Of course, this requires according functionality at that the peers processing the operator. The edit distance is very popular and suited to handle several common issues of Public Data Management applications, such as spelling errors. Throughout this work, we focus on this distance function for string data. Other popular string distance measures can be found in [147]. P-Grid respects data correlations in its storage model. This does not help to process string similarity. Even if we use an order-preserving hash function for strings that respects the lexicographical order $<_l$, this cannot reflect the edit distance. For instance, 'a' $<_l$ 'abc' $<_l$ 'd', but $edist$('a','d') $<$ $edist$('a','abc').

Evaluating String Similarity

We propose a q-gram-based approach. Q-grams are substrings of a fixed length q. A string s is split into q-grams by moving a sliding window over s and extracting q characters starting from each position. Among others, Navarro et al. [148] and Gravano et al. [77] propose methods for evaluating the edit distance using such q-grams. They developed their methods motivated by the high query times caused by very large data sets in centralized databases. The main observation of [148] is that for an edit distance of 1, the sets of q-grams from two strings will differ by at most q. Only these q contain the character affected by the one edit distance operation. The remaining q-grams *correspond* to each other. [77] introduced *count filtering*, *length filtering*, and *position filtering* as additional optimizations. These filters help to reduce the number of candidate q-grams and thus, the number of candidate strings that have to be checked finally. Count filtering is based on the fact that if $edist(s_1, s_2) \leq d$ is true, then s_1 and s_2 will share at least $(\max(|s_1|, |s_2|) + q - 1) - d \cdot q$ corresponding q-grams: when $s_1 = s_2$, they have exactly $\max(|s_1|, |s_2|) + q - 1$ common q-grams – and $d \cdot q$ is the maximum number of q-grams that can be affected by d edit distance operations. Length filtering utilizes that if $edist(s_1, s_2) \leq d$ is true, $|s_1|$ and $|s_2|$ cannot differ by more than d. Finally, for position filtering the q-grams are annotated with their starting position in the string. Two q-grams correspond to each other only if their positions differ by no more than d (if queried for distance d).

To exploit the benefits of q-grams, we build additional indexes, which can be used instead the standard indexes to answer string similarity queries. Rather than indexing whole strings, we split them into q-grams and index these q-grams. This is feasible for both, instance and schema level. For each triple t we use the hash keys $h(t[a]||qg_j(t[v]))$ for all (overlapping) q-grams qg_j of $t[v]$ to index the instance level. On schema level we use $h(qg_j(t[a]))$ for all (overlapping) q-grams qg_j of $t[a]$. Optionally, we can also index on pure instance level by using $h(qg_j(t[v]))$. This involves a non-negligible overhead depending on the actual choice of indexed attributes, but can be used to decrease query processing costs significantly.

As an example, consider a triple `(123 title mymovie)`. Building an index on instance level using all available 3-grams produces the following index items:

```
h(title||mym) → (123 title mymovie)
h(title||ymo) → (123 title mymovie)
h(title||mov) → (123 title mymovie)
h(title||ovi) → (123 title mymovie)
h(title||vie) → (123 title mymovie)
```

Also indexing on schema level results in the following additional index items:

```
h(tit) → (123 title mymovie)
h(itl) → (123 title mymovie)
h(tle) → (123 title mymovie)
```

If we use positional q-grams, we will index the schema level as follows:

```
h(0:tit) → (123 title mymovie)
h(1:itl) → (123 title mymovie)
h(2:tle) → (123 title mymovie)
```

Note that q-grams are only used for indexing, but are not stored in the triples explicitly. This is because we only utilize them to decrease routing efforts. If we would store q-grams or references rather than full strings, this would on the one hand decrease storage costs, but on the other hand demand for building or querying all candidate strings before finally computing the edit distance.

There exist versions of the edit distance assigning each operation costs ≥ 1 and allowing additional operations like transposition. This can be integrated straightforward. For instance, allowing transpositions as edit distance operation results in $2d + 1$ needed q-grams, because each operation can affect 2 q-grams [187].

In the following we assume a query for all strings similar to a provided query string. In centralized environments it is feasible to collect as many q-grams from candidate stings for comparison as possible, because this increases the total selectivity and results in less final candidates. This comprises all overlapping q-grams of all candidate strings. With the distributed index described above, each q-gram lookup results in a separate message. Thus, the number of queried q-grams should be minimized. With overlapping q-grams we have to query for $q \cdot d + 1$ q-grams, because each edit distance operation may affect q q-grams. A more efficient approach is to extract non-overlapping q-grams. Then, we only have to query for $d + 1$ q-grams, because none of them can be affected by the same operation. If we pick any $d + 1$ non-overlapping q-grams, at least one of them has to be fully contained in the comparison string. This conforms to the technique of querying a *q-sample* [199, 187]. We can save messages, but do not miss any candidates, on the costs of more candidates that have to be checked finally. Note that q-sampling can be applied only for the provided query string, but not when indexing inserted strings. As at least one of the queried q-grams has to be *fully* contained in a candidate string, we have to index all overlapping q-grams of all candidate strings. For q-sampling, we can, for instance, process the search string from left to right and construct $d + 1$ non-overlapping q-grams, starting from each q-th position. [148, 187] discuss interesting methods for estimating the selectivity of q-grams, which can be used to generate particularly good q-samples that result in only few candidates [187]. In contrast, extracting q-grams randomly will increase query-load balance and fairness, because popular queries are processed by different peers each time they are initiated.

A problem of the q-gram approach is its applicability for short strings. In order to make the q-gram approach work, a query string has to consist of enough characters to extract $d+1$ non-overlapping q-grams. With the method from above, we get at most $\lfloor \frac{n}{q} \rfloor$ q-grams from a string of length n. One option to extend this set is to add characters not included in the standard alphabet at the beginning and end of strings. We conceptually extend strings with q-1 leading occurrences of # and q-1 trailing occurrences of ‡. Indexing attribute name `title` as above with q=3 and positional q-grams would result in the following *additional* index items:

$h(-2{:}\#\#\mathtt{t}) \rightarrow (123\ \mathtt{title}\ \mathtt{mymovie})$
$h(-1{:}\#\mathtt{ti}) \rightarrow (123\ \mathtt{title}\ \mathtt{mymovie})$
$h(3{:}\mathtt{le}‡) \rightarrow (123\ \mathtt{title}\ \mathtt{mymovie})$
$h(4{:}\mathtt{e}‡‡) \rightarrow (123\ \mathtt{title}\ \mathtt{mymovie})$

Like this, we get $2 \cdot (q-1)$ q-grams more. Thus, we can extend the queried q-sample by two q-grams: one with leading # sign(s) and one with trailing ‡ sign(s). These additional q-grams have a decreased selectivity, but support to handle greater distances for short strings. Unfortunately, the decision on prefixing and suffixing strings has to be made before indexing the candidate strings. If we only extend the search string s, we might miss matching candidates c, because we conceptually generate an edit distance of $edist(s,c) + 2 \cdot (q-1)$ between s and c. A small constructed, but convenient, example illustrates this: imagine a query for all strings in an edit distance of $d \leq 2$ to `title` and a value of q=2. From `title`, we can construct only 2 non-overlapping 2-grams. We cannot just extend the query string to #`title`‡ and query for, e.g., the q-grams #`t`, `tl`, and `e`‡. This would miss the string `titel`, because we accidently chose those q-grams not contained in {`ti`, `it`, `te`, `el`}. But, it is in distance $d = 1$. If `titel` would have been extended before indexing as well, it would have produced the q-grams #`t` and `e`‡ as well. The length is mainly a problem for the search string, not for the indexed strings. As long as the length of each indexed string is greater or equal to q, the one fully contained q-gram that we look for will be indexed as well. Further, we may choose to use leading and trailing special characters for indexed strings, but do not use them in query strings. Note that falling back to overlapping q-grams does not help to overcome the issue of short strings either. If this would be feasible, then

$$\underbrace{\lfloor \frac{n}{q} \rfloor < d+1}_{\text{too less for q-sample}} \quad \wedge \quad \underbrace{n-q+1 \geq q \cdot d+1}_{\text{enough for overlapping q-grams}}$$

would have to be true. This is a contradiction, because

(I)	$\lfloor \frac{n}{q} \rfloor$	$<$	$d+1$		
(II)	$n-q+1$	$\geq$	$q \cdot d+1$		
(II)	$n-q$	$\geq$	$q \cdot d$		
(II)	n	$\geq$	$(d+1) \cdot q$		
(II) in (I)	$\lfloor \frac{(d+1) \cdot q}{q} \rfloor$	$<$	$\lfloor \frac{n}{q} \rfloor$	$<$	$d+1$
(I)	$\lfloor d+1 \rfloor$	$<$	$d+1$		

is obviously false as only integer values are allowed for d. If not enough q-grams can be extracted, we can only fall back to the implementations that materialize all attribute values and check similarity locally. To decrease the number of cases where this occurs, we suggest to use leading and trailing characters for strings that are expected to be rather short, such as the schema level of triples.

For q-gram indexing, the choice of hash function is very important. Since q-grams qg are rather short, generated hash keys $k = h(qg)$ are possibly too short. In this case, k would degenerate into a prefix key, which defines a range of responsible peers. For safety, when querying q-grams, we should be aware the fact that corresponding triples may reside on any peer p with $k \subset \Pi_p$. Thus, we have to include mechanisms to accordingly forward queries to all of these peers.

Conforming to [77], small values of q tend to give better results in combination with the introduced filters. But, selectivity of longer substrings is usually better than that of shorter substrings [187]. Without special characters, the number of index items $(n-q+1)$ decreases with increasing q. With special characters, the amount of index items $(n-q+1+2\cdot(q-1) = n+q-1)$ increases with increasing q. Another influence on the choice of q is the length of handled strings. If q is too large, we encounter problems with short strings. Following, choosing a value of q represents a trade-off between storage overhead, performance, and applicability. We do not discuss this issue further. We focus on q=3, because we observed that to be a good choice for most situations. This also conforms to theoretical considerations in [147] for approximate string matching with very long strings ($q = log_{|\Sigma|}(|s|)$, where s is a string on the alphabet Σ).

String Similarity Operators

Based on the q-gram indexes, we can implement special variants of routing operators. A great advantage of operators utilizing the q-gram index is that query plans are routed to at most $d+1$ peers, all the same how many candidate strings exist. String similarity extraction is the result of combining σ and ξ, string similarity materialization the result of combining σ and ω. We adhere to the notation already introduced and present $\sigma^{QG}(\xi)$ and $\sigma^{QG}(\omega)$. Both operators can be implemented as sequential operators (route to $d+1$ q-grams in a sequence) or as parallel operators (route to $d+1$ q-grams in parallel). To distinguish these variants, we write $\sigma^{QG,SEQ}$ and $\sigma^{QG,PAR}$. As the algorithms are very similar and can be deduced from the sequential and parallel approaches presented in Section 6.1, we illustrate only $\sigma^{QG,PAR}_{edist(?A,s)\leq d}(\xi_{['\mathtt{A}']})$ in Algorithm 5. Again, the optional parameter qg helps to distinguish the part that the initiator executes (lines 2 to 5) and the part that peers responsible for at least one queried q-gram execute (lines 8 to 13). For position filtering, we have to index positional q-grams rather than standard q-grams and apply the hash function accordingly. Length filtering can be applied in general, because we can always compare the length of strings before computing their edit distance. We include this in line 8. Count filtering is implicitly applied by picking $d+1$ non-overlapping q-grams.

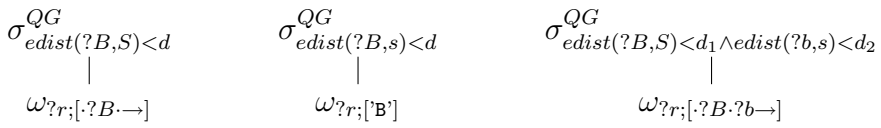
Q-gram-based operators are conceptually very similar to hash-based operators. Whereas ω^{HASH} uses triple sets from the output of other operators, σ^{QG} uses the q-grams generated from a provided search string. Thus, similar optimizations should be integrated. The parallel variants will usually produce duplicated results, because q-grams queried in parallel may belong to the same candidate string. This can be prevented during query time only when using the sequential approach. Note that a sequence length of $d+1$ is rather short, which decreases the risk of broken chains. Besides checking for duplicates, sequential operators should process *all* q-grams that the current peer is responsible for. This can lower the number of forwarded query plans and thus, the number of generated messages.

The number of queried q-grams represents a trade-off between the number of generated messages, local costs, and robustness. By querying the minimal number of $d+1$ q-grams, we optimize the main objective, which is to minimize the number of messages. The more q-grams we query, the less final candidates we will produce. We can filter out all candidates that have too much non-matching q-grams – if we query for all q-grams of the search string, we can filter out also all candidates that do not have enough matching q-grams. But, such extended count

Algorithm 5: Q-gram-based similarity extraction: $\sigma^{QG,PAR}_{edist(?A,s)\leq d}(\xi_{['A']})$

```
   Input: attribute name A, string s, distance d[, q-gram qg]
 1 if qg = NULL then
 2 |   determine d + 1 q-grams Q from s;
 3 |   forall qg ∈ Q do
 4 |   |   forward(h(A||qg), qg);
 5 |   end
 6 end
 7 else
 8 |   forall t ∈ OT_local(t[a] = A ∧ ||t[v]|| − |s|| ≤ d) do
 9 |   |   if edist(t[v], s) ≤ d then
10 |   |   |   add-to-result({t});
11 |   |   end
12 |   end
13 |   signalize(DONE);
14 end
```

filtering requires to use the sequential operator or synchronization peers. Further, if we query only $d+1$ q-grams, we might miss candidates if one of the responsible peers fails. This can be prevented by applying redundancy on query level through querying more than $d+1$ q-grams using the parallel operator.



(a) Schema level (b) Instance level (c) Schema and instance level

Figure 6.3: Similarity materialization on different levels

We illustrate three examples for combining σ and ω in Figure 6.3. On schema level (Figure 6.3(a)) we materialize all triples belonging to attributes with a name similar to a provided query string S. This is achieved by looking up q-grams qg_i of S using $h(qg_i)$. On instance level (Figure 6.3(b)) we materialize triples belonging to an attribute B only if the attribute value is similar to a provided query string s. Thus, we look up the keys $h(\mathtt{B}||qg_j)$ for q-grams qg_j of s. This can be combined to form a similarity materialization on both levels as shown in Figure 6.3(c). Here, we route for the q-grams qg_i of the attribute name S using $h(qg_i)$. Similarity between attribute values $?b$ and s is checked only locally. Note that similarity on OID level is possible as well. But, following the operator definitions from Section 5.1, this represents a similarity join rather than a materialization. ω always combines triples on equal OIDs. Besides possible performance benefits (all input triple sets can be processed with only $d+1$ messages), $\sigma^{QG}(\omega)$ provides an alternative to ω^{HASH} for processing unbound triple patterns. We get back to this issue below after introducing similarity join implementations.

A third operator that can be implemented based on q-grams is the similarity join operator. A similarity join is one of the most important similarity operators, because it is a powerful tool to overcome heterogeneity at different levels [55]. It allows to handle semantic inconsistencies for schema integration and to address inconsistencies and inaccuracies in the data. The according physical operators represent a combination of the hashing approach and the q-gram

approach. The left side input T_l provides the values for the join. From each of these values we extract at least $d+1$ q-grams. These $|T_l| \cdot (d+1)$ or more q-grams are queried in parallel or in sequence. As the input strings can also be processed in sequence or in parallel, this results in a doubled choice between sequential and parallel processing. We denote the operators accordingly by $\bowtie^{HASH,SEQ/PAR}$ ($\xi^{QG,SEQ/PAR}$). We illustrate the q-gram-based implementation in Algorithm 6.

Algorithm 6: Q-gram-based similarity join: $\bowtie^{HASH,PAR}_{edist(?A,?B)\leq d}$ ($\xi^{QG,PAR}_{['B']}$)

Input: variable name $?A$, attribute name B, left side input T, distance d[, attribute value v_A, triple set ts]

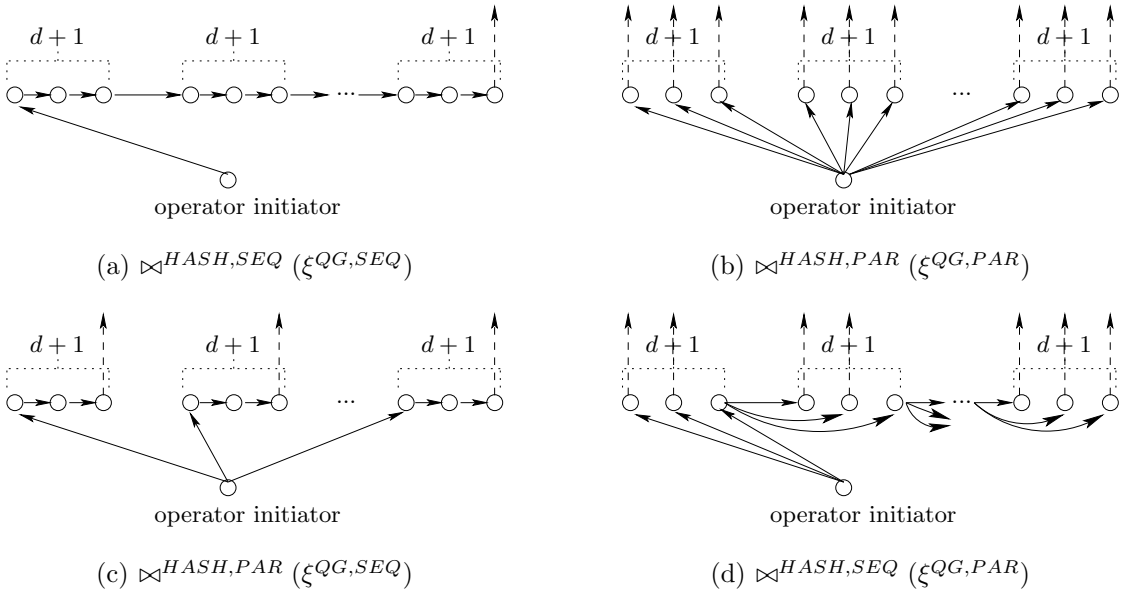
```
1  if v_A = NULL then
2      forall ts ∈ T do
3          v_A = ref(?A, ts);
4          determine d + 1 q-grams Q from v_A;
5          forall qg ∈ Q do
6              forward(h(B||qg), v_A, ts);
7          end
8      end
9  end
10 else
11     forall t ∈ OT_local(t[a] = B ∧ ||t[v]|| − |v_A|| ≤ d) do
12         if edist(t[v], v_A) ≤ d then
13             add-to-result-tuple(ts ∪ {t});
14         end
15     end
16     signalize(DONE);
17 end
```

In analogy to hashing-based operators, several optimizations should be integrated into Algorithm 6. Equal strings and q-grams from the left side should be queried together. Similar, using sequential implementations can help to reduce the number of generated messages, but bears the risk of delays due to broken sequences. Figure 6.4 illustrates the routing schemes of the different variants of the q-gram-based $\bowtie^{HASH}$ operator. In analogy to the similarity materialization, similarity joins can be processed on arbitrary combinations of schema level and instance level – and even of OID level. Note that the q-gram index can be used also to implement equi-joins. This provides an alternative to using the attribute-value index.

All q-gram operators introduced in this section can be used only as a combination of σ or $\bowtie$ with one direct child ξ or ω. This has to be regarded for the process of query optimization.

Evaluating Similarity on Numerical Data

For similarity queries on numerical attributes we map the provided distance to a corresponding interval and process them as range queries. But, to support this efficiently on the basis of range queries, the used DHT has to support order preservation. P-Grid supports this and also deals with the possible imbalance problems efficiently (see Section 4.1.1). The implementation of



(a) $\bowtie^{HASH,SEQ}$ ($\xi^{QG,SEQ}$) (b) $\bowtie^{HASH,PAR}$ ($\xi^{QG,PAR}$)

(c) $\bowtie^{HASH,PAR}$ ($\xi^{QG,SEQ}$) (d) $\bowtie^{HASH,SEQ}$ ($\xi^{QG,PAR}$)

Figure 6.4: Routing schemes for $\bowtie^{HASH}$ with $d = 2$

such similarity queries corresponds to that of nearest neighbor queries, which are introduced for string and numerical data in the next section.

Now, after introducing implementations for standard routing operators as well as similarity operators, we can get back to the problem of unbound triple patterns. As mentioned in Section 5.1, queries containing such patterns should be rejected if they require to lookup all triples managed in the system. Meanwhile, we presented several operator implementations that utilize an implicit binding of variables. Backed by the set of physical operators, we can adhere to the following rules:

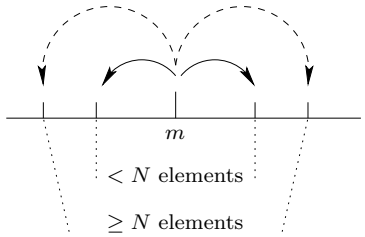
1. If a query results in an unbound ξ on leaf level, the query will be rejected in principle.

2. As an exception to rule 1, selections can provide bindings for unbound ξ on leaf level. A similarity selection combined with an unbound extraction is supported by a similarity extraction $\sigma^{QG}(\xi)$. Range predicates are supported by a range-query extraction $\sigma^{RQ}(\xi)$.

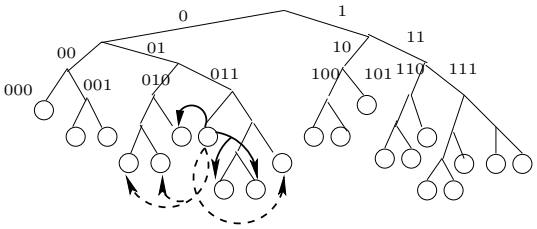
3. Rule 2 holds in analogy for unbound triple patterns that can be represented by a materialization ω. Even if no according selection to form $\sigma^{QG}(\omega)$ or $\sigma^{RQ}(\omega)$ exists, hash materialization ω^{HASH} can be applied.

4. As a last exception to rule 1, equi-joins where only one side is unbound are supported by $\bowtie^{HASH}$ (ξ/ω). Similarity joins of this class are supported by $\bowtie^{HASH}$ (ξ^{QG}/ω^{QG}).

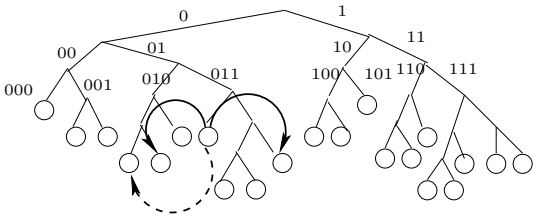
5. All rules can be applied only if the required indexes exist. Otherwise, according queries have to be rejected.



(a) General principle



(b) Numerical data



(c) String data

Figure 6.5: Principle of φ

6.2.2 Ranking Operators

In this section, we present alternative implementations for the ranking operators φ and Φ. Ranking queries, particularly nearest neighbor queries, are also regarded as a kind of similarity queries. We already discussed the local implementations of ranking operators in Section 6.1. The physical operators presented next implement sophisticated algorithms specifically designed for the according ranking process. In contrast to the similarity operators from Section 6.2.1, the following operators use only the standard indexes introduced in Section 4.2.1.

Top-N Operator

We begin with the top-N operator φ. For algorithmic details, we base our explanations on the case of $\varphi_{\texttt{NN},m,0;?A;N}(\xi_{\texttt{'A'}})$. Following the definition from Section 5.1, this corresponds to a N nearest neighbor query ($N > 0 \wedge d \leq 0$). The operator extracts the N triples belonging to attribute `A` that are closest to the query value m. Which distance measure is applied depends on the type of input. The following approach is also suited for all other cases of operator parameters. We briefly highlight the required minor modifications below. The local operator φ^{LOC} and the synchronizing operator φ^{SYNC} require that all input triples are extracted by preceding operators. In most cases, φ will be placed directly above a routing operator like $\xi_{['A']}$. Similar to other operator combinations, we can combine φ with such a routing operator, which results in the implementation presented here. It is based on the idea of guessing a range in which the N highest ranked triple sets are contained before querying it. Query plans are sent to only the peers responsible for this range. If not enough triple sets are found in the guessed range, we have to expand it and re-initiate according queries. This is not a novel idea. [40] discusses how to apply this approach in relational databases. In our case, for numerical data and the Euclidean distance, we can directly map the estimated range to range queries provided by the DHT. This leverages efficiency only when the DHT supports order-preserving hashing, as P-Grid does, or other methods for accelerating range query processing. Range queries are not suited for string data. But, we can utilize the q-gram index to query for strings in a certain

distance. The distance again is estimated by guessing an according range before querying. In Figure 6.5 we illustrate the general principle. We assume $h(\texttt{A}) = 01$ and that we have to initiate two rounds of range queries. The first one is referred by the solid lines, the second one by dashed lines. For strings, we query for $d = 1$ in the first round and for $d = 2$ in the second round. With a smart algorithm that uses a kind of prefetching as presented below, we have to query only one q-gram more in the second round.

Algorithm 7: Top-N query: $\varphi^{PAR}_{\texttt{NN},m,0;?A;N}(\xi_{['\texttt{A}']})$ – initial phase

Input: attribute name A, limit N, value m
$R = \{t \in OT_{local} | t[a] = \texttt{A}\}$;
$min = \min_{t \in R} t[v]$; $max = \max_{t \in R} t[v]$;
low = min-val(Π_{local}, A); $high$ = max-val(Π_{local}, A);
/* can be checked locally */
if $h(low)$ = first-key(A) **then** $low = -\infty$; **end**
if $h(high)$ = last-key(A) **then** $high = \infty$; **end**
/* determine the local density for attribute A */
$dense = \frac{\max(1,|R|)}{high - low + 1}$; /* if $high = \infty$, use max instead, min respectively */
if $|R| < N$ **then**
  $r = \frac{N}{dense}$; /* estimate a range r in which N results could be */
  $d_1 = d_2 = \lceil \frac{r}{2} \rceil$;
  if $high = \infty$ **then** $d_1 = \lceil r - (max - m) \rceil$; **end**
  $fr_1 = m - d_1$; $to_1 = low - \varepsilon$;
  if $low = -\infty$ **then** $d_2 = \lceil r - (m - min) \rceil$; **end**
  $fr_2 = high + \varepsilon$; $to_2 = m + d_2$;
  /* less than N elements for attribute A */
  if $fr_1 \geq to_1 \wedge fr_2 \geq to_2$ **then**
    sort(R, m);
    signalize($DONE$);
  end
  else
    if $fr_1 < to_1$ **then** range-forward($h(\texttt{A}||fr_1), h(\texttt{A}||to_1)$,top-$N$-collect(A, N, m, d_1, R)); **end**
    if $fr_2 < to_2$ **then** range-forward($h(\texttt{A}||fr_2), h(\texttt{A}||to_2)$,top-$N$-collect(A, N, m, d_2, R)); **end**
    forward(p_{sync},top-N-sync(A, $N, m, R, \emptyset, fr_1, to_2$));
  end
end
else
  limit(sort(R, m), N);
  signalize($DONE$);
end

Range queries on the Euclidean distance and similarity queries on the edit distance can be processed in sequence or in parallel. This results in two physical operators, namely φ^{SEQ} and φ^{PAR}. We first discuss the algorithm for numerical data and afterwards explain how it can be mapped to string data. For convenience, we split the algorithm into three parts. The first peer that processes $\varphi^{SEQ/PAR}$ routes the query to the peer p_m responsible for m. On this peer, the actual processing starts as shown in Algorithm 7. The idea is to deduce from the number of locally available index items and the range of local responsibility to the guessed range to query. This assumes load balancing between the peers, which P-Grid tries to achieve. First, the local minimal and maximal values are determined from the set of local triples OT and stored in min and max. Usually, these existing values will differ from the lowest and highest values that p_m is theoretically responsible for. Thus, we determine these bounds separately and store

them in *low* and *high*. `min-val(`Π`,A)` returns the minimal value for attribute `A` that the peer with path Π is responsible for. `max-val()` returns the corresponding maximum, respectively. *low* and *high* are used to identify the range that was already queried – in the initial phase, this is the local range of the peer responsible for $h(m)$. To achieve a terminating algorithm, we have to check if p_m is responsible for the minimal key of attribute `A`, the maximal key, or both. This is considered accordingly by setting $low = -\infty$, $high = \infty$ respectively. If p_m is the only peer responsible for values from `A`, $low = -\infty \wedge high = \infty$ will be true. After determining the local range, it is used to compute the local density of values, which is stored in *dense*. For this, we use the number of already extracted triples, i.e., in the initial phase the number of corresponding local triples. If we found less than N elements, we will have to include more peers into processing of the operator. If we already found N values or more, we can sort that set of candidates, prune it to N, and finish processing of the operator (lines 25 to 26). Based on the assumption that the local density conforms approximately to the density at neighboring peers, we use *dense* to guess a range r to query. We use the range from $m - \lceil \frac{r}{2} \rceil$ to $m + \lceil \frac{r}{2} \rceil$. Respecting the already queried range, this results in two subranges that can be queried independently. If the minimal value of `A`, the maximal value, or both are included in the already queried range, the range will be shifted accordingly. If only one of the extrema is contained, the range will be shifted to the direction that is not covered completely. The values of *low* and *high* are used for that purpose (line 10 and line 12). Consequently, only one subrange will be queried. The range queries are initiated in lines 19 and 20. Using φ^{PAR}, this is done by calling `range-forward()`. This method takes an additional parameter to indicate the next phase to process (`top-`N`-collect()`). If both extrema are contained, fr_1 will be greater than to_1 and fr_2 will be greater than to_2 (again, due to the values of *low* and *high*). This will result in stopping processing in line 16. For the unlikely case that p_m holds no value of `A` at all, we assume $|R| = 1$ in line 6. A very special case is that no value for the queried attribute is available in the whole system. This is even more unlikely, but possible. Only in this case, *dense* would evaluate to 0 in line 6, which can be handled accordingly. Finally, line 21 is used to inform the synchronization peer about the new queried range.

Algorithm 8: Collection phase top-N-collect() used in $\varphi^{PAR}_{\texttt{NN},m,0;?A;N}(\xi_{['\texttt{A}']})$

Input: attribute name `A`, limit N, value m, distance d, intermediate result R

1 $R = R \cup \{t \in OT_{local} | t[a] = \texttt{A} \wedge m - t[v] \leq d\}$;
2 **if** $|R| < N$ **then** $C = \text{limit}(\text{sort}(\{t \in OT_{local} \setminus R | t[a] = \texttt{A}\}, m), N - |R|)$; **end**
3 **else** $R = \text{limit}(\text{sort}(R, m), N)$; **end**
4 $low = \text{min-val}(\Pi_{local}, \texttt{A})$; $high = \text{max-val}(\Pi_{local}, \texttt{A})$;
`/* can be checked locally */`
5 **if** $h(low) = \text{first-key}(\texttt{A})$ **then** $low = -\infty$; **end**
6 **if** $h(high) = \text{last-key}(\texttt{A})$ **then** $high = \infty$; **end**
7 forward(p_{sync},top-N-sync($\texttt{A}, N, m, R, C, low, high$));

If the assumption on load balancing holds, we will have to process only the ranges determined in the initial phase. Each peer that receives one of the range queries executes `top-`N`-collect()` as shown in Algorithm 8. Local values that are in the queried distance are added to the set R of result values. Additionally, if $|R|$ is still lower than N, up to $N - |R|$ candidate values are stored into C. This applies the aforementioned prefetching strategy. Like this, we can prevent to query peers multiple times if we have to re-initiate range queries. As in Algorithm 7, the peers determine the bounds of the local responsibility in *low* and *high*. Finally, the current results, candidates, and bounds are sent to a synchronization peer.

Algorithm 9: Synchronization phase top-N-sync() used in $\varphi^{PAR}_{\texttt{NN},m,0;?A;N}(\xi_{['\texttt{A}']})$

Input: attribute name A, limit N, value m, intermediate result R, candidates C, lower bound low, upper bound $high$

```
1  if |R| < N then
2      R = limit(sort(R ∪ C,m),N) ;                                  /* use candidates */
3      min = min_{t∈R} t[v]; max = max_{t∈R} t[v] ;                  /* range of found values */
4      if |R| < N then
5          dense = max(1,|R|)/(high−low+1) ;   /* if high = ∞, use max instead, min respectively */
6          r = N/dense ;                       /* estimate a range r in which N results could be */
7          d_1 = d_2 = ⌈r/2⌉;
8          if high = ∞ then d_1 = ⌈r − (max − m)⌉; end
9          if low = −∞ then d_2 = ⌈r − (m − min)⌉; end
10     end
11     else
12         if max − m > m − low then
13             d_1 = max − m; d_2 = 0 ;        /* dist to found max > dist to covered low */
14         end
15         else if m − min > high − m then
16             d_1 = 0; d_2 = m − min ;        /* dist to found min > dist to covered high */
17         end
18         else d_1 = d_2 = 0; end
19     end
20     fr_1 = m − d_1; to_1 = low − ε;
21     fr_2 = high + ε; to_2 = m + d_2;
22     if fr_1 ≥ to_1 ∧ fr_2 ≥ to_2 then
23         R = limit(sort(R, m), N) ;   /* probably less than N values for attribute A */
24         signalize(DONE);
25     end
26     else
27         if fr_1 < to_1 then range-forward(h(A||fr_1), h(A||to_1),top-N-collect(A, N, m, d_1, R)); end
28         if fr_2 < to_2 then range-forward(h(A||fr_2), h(A||to_2),top-N-collect(A, N, m, d_2, R)); end
29         forward(p_sync,top-N-sync(A, N, m, R, ∅, fr_1, to_2));
30     end
31 end
32 else
33     R = limit(sort(R, m), N) ;                /* we found N items in the queried range */
34     signalize(DONE);
35 end
```

The synchronization peer acts as illustrated in Algorithm 9. We omit the process of collecting input query plans from the peers responsible for the queried ranges. In this process, the single sets R and C are merged, low is set to the minimal queried value, and $high$ is set to the maximal queried value over all subqueries. In the initial phase, the queried subranges are signalized by an according message in line 21 of Algorithm 7. In conjunction with the interval bounds that each peer sends together with R and C, the synchronization peer can decide whether still some range-query results are missing or not. In Section 7.2 we introduce an extended approach for that task. After collecting all subqueries, the synchronization phase is very similar to the initial phase. If $|R| \geq N$, we can finish processing. Otherwise, we first try to extend R using the candidates from C. If still less than N triple sets are received, we compute a new range r to query, this time using the density of the current result. Then, the collection and synchronization phases are started again. But, even if we received N or more triple sets in $R \cup C$, we might have queried an asymmetric interval. In this case, we have to issue one compensating range query in order not to miss relevant values. The resulting range to query is determined in lines 12 to 17. Below we present an example to illustrate this issue. We always use an ε value for determining the new subranges. Since all range queries are processed on closed intervals, we use that small offset to prevent querying the same values twice. For integer values ε is always 1, for floating point values it should be sufficiently small. Apart from that, processing of floating point values adheres to the same principle. We have to determine the density using $dense = \frac{\max(1,|R|)}{high-low}$ and do not have to use rounding when determining the subranges. Further modifications are not needed.

The synchronization peer can be freely chosen in each round. It should be a peer that has enough resources and processing capabilities and that is reliable. In the sequential variant, the "synchronization" peer will always be the last one in the currently queried range. The two subranges can be queried one after another or they can be queried independently. This allows for implementing a hybrid form between φ^{SEQ} and φ^{PAR}. If the subranges are queried independently, a synchronization peer will be required even if each range is processed sequentially. As in all synchronizing operators, the synchronization peer has to be defined in each round before sending any plan copy. If the query initiator has enough resources, it will be reasonable to choose it as the synchronization peer, because it is expected to be online. All the same which peer is chosen, it has to be identified by an unique ID, such as its IP address. Even if the query initiator synchronizes in each round, this is still conforms to the stateless fire-and-forget strategy. The responsible peer processes a "new" operator each time range queries are returned for synchronization. Nothing has to be stored locally, all needed information is encapsulated in the traveling query plans. Of course, this can be combined with a revision-based online processing as introduced for other synchronizing operators.

The handling of `MIN` and `MAX` as ranking functions is intuitive. We first route to $m = min$ or $m = max$ using `first-key(A)`, `last-key(A)` respectively. Then, we can use the same algorithm. The minimal or maximal value provides the border of the possible range, which is detected and processed accordingly. Thus, for the `MAX` ranking function we start at the upper bound of the attribute's range and proceed by shifting the search interval downwards. Other cases of operator parameters are also handled by the presented algorithm. If $N > 0 \wedge d > 0$, processing will be the same, but the maximal queried distance will be limited by d. If $N \leq 0 \wedge d > 0$, this will correspond to a single range query or similarity query with distance d. Finally, if $N \leq 0 \wedge d \leq 0$, all according triples have to be extracted and sorted, but neither pruned nor limited by a maximal distance d. In this case, local or synchronizing operators should be used instead of the physical operators from this section.

For convenience, the algorithms are presented on the assumption of only positive values. Modifications for supporting also negative values are straightforward. Further, in the algorithms

we can replace the assumption of a uniform distribution by using other distribution models. Even histograms can be used for handling arbitrary distributions. As a final extension, the approach can be combined with probabilistic guarantees. Then, the determined query intervals depend on a distribution model in combination with a desired probability of accuracy (with $x\%$ we missed at most y of the top items) and a probability of re-initiating the range querying. We proposed this before for PDMS in [93].

A small example illustrates the different cases that we consider in the algorithms. We assume five peers responsible for values of an integer attribute `A` as follows. Each peer p_i is responsible for the range $from \rightarrow to$ and holds the values $\{v_1, \ldots, v_{n_i}\}$:

- $p_1 : 10 \rightarrow 14; \{11, 13, 14\}$
- $p_2 : 15 \rightarrow 17; \{15, 17\}$
- $p_3 : 18 \rightarrow 21; \{19, 20, 21\}$
- $p_4 : 22 \rightarrow 27; \{22, 23\}$
- $p_5 : 28 \rightarrow 30; \{28, 30\}$

The operator $\varphi_{\texttt{NN},20,0;?A;10}(\xi_{[\texttt{'A'}]})$ would be processed as follows:

1. First, a corresponding query plan is routed to p_3, because this peer is responsible for $m = 20$. At p_3, we find three according values and determine a local density $dense = \frac{3}{21-18+1} = 0.75$. Thus, we compute the range $r = \frac{10}{0.75} = 13.\overline{3}$. Two range queries are issued, the first q_1 spanning from $m - d_1 = 20 - \lceil \frac{r}{2} \rceil = 13$ to 17, the second q_2 from 22 to $m + d_2 = 20 + \lceil \frac{r}{2} \rceil = 27$.

2. q_1 is sent to p_1 and p_2. p_2 includes all its values into R. p_3 includes 13 and 14 into R and 11 as a candidate into C.

3. q_2 is sent to p_4, which includes 22 and 23 into R.

4. At the synchronization peer, we receive $R = \{13, 14, 15, 17, 19, 20, 21, 22, 23\}, C = \{11\}$, $low = 10$, and $high = 27$. Since $|R| = 9 < 10$, we use the candidates as well. The distance between m and the current minimum is $m - min = 20 - 11 = 9$. This is greater than the distance between m and $high$, which is $high - m = 27 - 20 = 7$. Thus, we issue a compensating query q_3 for the distance $d_2 = m - min = 9$, which spans from 28 to 29.

5. q_3 is sent to p_5, which inserts 28 into R.

6. In the final synchronization step, $R = \{11, 13, 14, 15, 17, 19, 20, 21, 22, 23, 28\}$ is pruned to $R = \{13, 14, 15, 17, 19, 20, 21, 22, 23, 28\}$ and the operator signalizes $DONE$.

The following special cases are also covered:

1. If p_1 does not hold 11, we will compute $dense = \frac{9}{27-10+1} = 0.5$ in step 4. This results in a range $r = \frac{10}{0.5} = 20$. Thus, we determine the next ranges to query spanning from 10 to 9 and from 28 to 30. The first range is already covered. Thus, we issue one range query q_3 from 28 to 30.

2. If we do not include candidates (e.g., in step 2), we will have to use $dense = \frac{9}{27-13+1} = 0.6$ in step 4. The resulting range is $r = \frac{10}{0.6} = 16.\overline{6}$. Thus, we query from 11 to 12 and from 28 to 29, which involves q_1 unnecessarily again.

3. If p_5's responsibility starts at 27, it will be hit by q_2. Then, p_5 includes all its values into C, which stops the processing already in step 4.

4. If p_3 is the only peer responsible for values from `A`, it will determine the density $dense = \frac{3}{21-19+1} = 3$, but issues no further queries, because $low = -\infty$ and $high = \infty$.

5. If the maximum value of `A` is 21, p_3 will compute the density $dense = 0.75$ and a range $r = 13.\overline{3}$ as in step 1 above. But, it issues only a subquery for the range from $m - d_1 = 20 - (\lceil r - (21 - 20) \rceil) = 7$ to 17. This also indicates how queries for `MIN` or `MAX` rather than `NN` are handled.

Algorithm 10: Top-N query on strings: $\varphi^{PAR}_{\texttt{NN},m,0;?A;N}(\xi_{['\texttt{A}']})$ – initial phase with sketched synchronization phase

Input: attribute name `A`, limit N, value m
$R = \{t \in OT_{local} | t[a] = \texttt{A}\}$;
`/* determine the local density for attribute A */`
$dense = \frac{\max(1,|R|)}{\max_{ts \in R} edist(m,ts)}$;
$P = \{\Pi_{local}\}$;
if $|R| < N$ **then**
  $d = r = \frac{N}{dense}$; `/* estimate a range` r `in which` N `results could be */`
  if whole range of `A` is covered by P **then**
    limit(sort(R, m), N);
    signalize($DONE$);
  end
  `/*` Q `not needed in initializing phase, but in synchronizing phase */`
  $Q = \emptyset$;
  determine $d + 1 - |Q|$ q-grams Q' from m not in Q;
  forall $qg \in Q'$ **do**
    if $\nexists p \in P(p \subseteq h(\texttt{A}||qg))$ **then** forward($h(\texttt{A}||qg)$,top-N-collect(`A`, N, m, d, R)); **end**
  end
  forward(p_{sync},top-N-sync-str(`A`, $N, m, R, \emptyset, Q \cup Q', P$));
end
else
  limit(sort(R, m), N);
  signalize($DONE$);
end

The `MIN` and `MAX` ranking functions are also applicable to strings. But, we expect these combinations to be rather unlikely. Thus, we consider only the `NN` ranking function for top-N queries on string data. As in the numerical case, the guessed range can be mapped to a distance value. But, while this corresponds to a connected range of neighboring peers in the numerical case, the strings similar to m will be uniformly distributed over all peers. This is due to the characteristics of the edit distance. Fortunately, the q-gram-based operators efficiently support string similarity queries. The q-gram index is the only choice we have for this approach. Using the attribute-value index would have to involve all peers responsible for attribute `A` in the first round. This would stop processing after the first "round", but probably involve a large number of peers. The interval-based approach in conjunction with q-grams promises to be more efficient, assuming that we usually require a small number of rounds. The first peer(s) that starts processing of $\varphi^{QG,SEQ/PAR}$ does not have to route a query plan to p_m initially. Due to the non-order-preserving hashing of strings, this is not needed. But, it is mandatory that the query

plan is sent to a peer responsible for a part of the triples belonging to `A`. Otherwise, we could not determine a local density of according values. The main differences between both cases can be deduced from Algorithm 10, which summarizes the processing for strings. We describe the initial phase, but also sketch specific steps of the synchronization phase. The algorithm is simpler, because we do not have to consider maximum and minimum. They do not exist for the edit distance. But, to guarantee termination of the algorithm, we still have to check if the whole range of `A` is already covered. This can be achieved only by maintaining a set of all paths of peers that already processed the operator. In the algorithm, this is indicated by the set P, which can be shipped together with the query plan. This is in analogy to recording *low* and *high* in the numerical case. Instead of max, min, low an *high*, we use only the queried distance d. Still, each involved peer includes all matching candidates into C. This has a valuable effect. If similarity queries are re-initiated in a second or later round, the queried distance will be increased by Δ. This results in querying Δ more q-grams compared to the preceding round. The q-grams used before may be used again, which is not needed if the responsible peers included all candidates before. Thus, only Δ q-grams have to be queried. This corresponds to Figure 6.5(c). Further, if one of the Δ new q-grams qg maps to a peer p that has already been involved, i.e., $\exists p \in P(\Pi_p \subset h(qg))$, qg can be removed from the list as well without requiring replacement. To achieve this, P is also used to prevent that peers are queried twice. Q and also P are used to sketch the processing of the synchronization phase. Further modifications of the collection and synchronization phases are intuitive.

If the top-N operator supports string and numerical data, we can use it for schema level, instance level, and OID level. For numerical values (instance or OID level) this requires order-preserving hashing. Processing top-N operators on OIDs can be useful, e.g., when issued on semantic data. This again highlights the importance of mechanisms to detect the number and kind of actually available indexes.

The algorithms from this section are suited only for distance measures and ranking functions that can be mapped to either range queries or similarity queries. For multidimensional top-N queries, the primary ranking function (the first in the list *rank*) is used to extract according triples. The remaining functions can be evaluated on the gathered data locally at the synchronization peer. Thus, this also supports arbitrary distance measures and ranking functions for the secondary variables and beyond. For other cases, e.g., one single ranking function over multiple attributes, φ^{LOC} has to be used. The local operator supports arbitrary rankings in all dimensions. Note that in all cases the user still has to specify whether to minimize, maximize, or determine nearest neighbors.

Skyline Operator

Ranking queries are a prerequisite for handling extremely large and heterogeneous data collections in a meaningful way. While top-N queries on multiple attributes are very useful, they bear the disadvantage that a priority has to be defined a priori on the used ranking functions. This is often not feasible. As mentioned in Section 5.1, an alternative for multidimensional ranking are skyline queries [33], which determine best possible trade-offs between all equally weighted ranking goals.

A main objective for skyline queries in distributed systems should be to minimize the search space. Since all data points are distributed among several peers, this will minimize the number of involved peers. Thus, the number of messages will be decreased. There are several conceivable approaches for narrowing the search space. For instance, [208] achieves this by first contacting the peer responsible for the origin (if only `MIN` is used in all dimensions) of the skyline space. At this peer, the point dominating the largest region is determined and used for pruning the

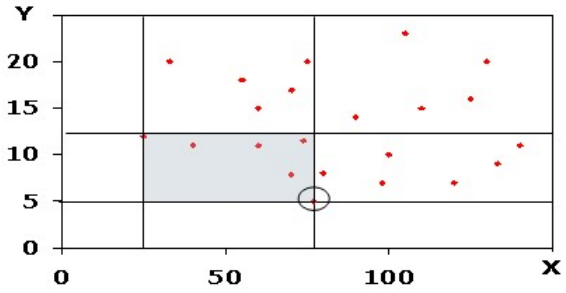


Figure 6.6: Frame skyline approach

remaining search space. We propose the *frame skyline* approach. We focus on `MAX` and `MIN` based on the Euclidean distance as ranking functions and thus, on numerical data. In the context of skylines queries, we expect these ranking functions to be the most popular ones. In the following, we exemplarily consider a two-dimensional skyline query with a `MIN` ranking in each dimension. We narrow the search space as follows. Elements from the data set that are minimal in one dimension are definitely part of the skyline – they cannot be dominated by any other item. In parallel, these elements narrow the search space for the second attribute. Figure 6.6 illustrates this. The point minimal in dimension y (marked with a circle) provides the maximal value for dimension x we have to consider. Any other point revealing a larger x-value cannot be part of the skyline, because it is definitely dominated in both dimensions. Based on the extrema of all dimensions, we can thereby narrow the search space as illustrated by the shaded frame. Using the approach from [208], we would have to identify the peer responsible for the point closest to the origin. There is no single peer that the origin is directly assigned to, because both dimensions are indexed separately. But, even if we find the closest point – requiring another probably expensive operator – the search space is only narrowed by one point. This can result in a larger search space that has to be considered. In Figure 6.6, the point minimal in dimension x would be used. Narrowing using both extrema is much more efficient here.

Algorithm 11: Initialization for $\Phi_{\texttt{MIN},\ldots,\texttt{MIN};?A_1,\ldots,?A_n}(\omega_{?A_i;[\texttt{A}_1],\ldots,[\texttt{A}_{i-1}],[\texttt{A}_{i+1}],\ldots,[\texttt{A}_n]}(\xi_{[\texttt{A}_i]}))$

Input: attribute names $\texttt{A}_1 \ldots \texttt{A}_n$
1 **forall** $\texttt{A}_i$ **do**
  `/* find min of` $\texttt{A}_i$ `and materialize all` $\texttt{A}_j, j \neq i$ `*/`
2 $SQ = \text{sub-plan}(\omega_{[\texttt{A}_j]\forall j \neq i \wedge j < n}(\varphi_{MIN;?A_i;1}(\xi_{[\texttt{A}_i]})))$;
3 add-synchronization-root(SQ,query-frame);
4 forward(init(SQ));
5 **end**

To compute skylines following the frame skyline approach, we can make direct use of the DHT's efficient query processing capabilities. We contact only those peers responsible for elements included in the resulting frame using range query mechanisms. We developed three basic variants: sequential Φ^{SEQ}, parallel Φ^{PAR}, and hybrid Φ^{HYB}. The full algorithmic details are rather complex. Since they differ only in details, we do not list them for all variants. Rather, we present one general algorithm illustrating all three variants. Similar to the top-N operator, the skyline operator is processed in phases, which are shown in Algorithms 11 to 14. We follow the same fire-and-forget strategy using freely chosen synchronization peers when

needed. In analogy, the variants presented here are only applicable when a skyline operator can be combined with routing operators like ξ. Full algorithmic details, including implementation issues and cost estimations, can be found in [142].

Algorithm 12: `query-frame()` for $\Phi_{\texttt{MIN},\ldots,\texttt{MIN};?A_1,\ldots,?A_n}(\omega_{?A_i;[\texttt{A}_1],\ldots,[\texttt{A}_{i-1}],[\texttt{A}_{i+1}],\ldots,[\texttt{A}_n]}(\xi_{[\texttt{A}_i]}))$

Input: attribute names $\texttt{A}_1 \ldots \texttt{A}_n$
1 $frame = \emptyset$; $i = 1$;
2 **while** $i \leq n$ **do**
3 $\quad frame = frame \cup reply_i$; `/* collect results in synchronizing phase */`
4 $\quad i = i + 1$;
5 **end**
6 **forall** $\texttt{A}_i$ **do**
7 $\quad min_i = \min_{t \in frame(t[a]=\texttt{A}_i)} t[v]$;
8 $\quad max_i = \max_{t \in frame(t[a]=\texttt{A}_i)} t[v]$;
9 **end**
10 **if** $\Phi^{PAR} \vee \Phi^{HYB}$ **then**
11 $\quad r = \arg\min_{i \leq n}(max_i - min_i)$;
12 $\quad$ range-forward$(h(\texttt{A}||min_r), h(\texttt{A}||max_r))$;
13 **end**
14 **else**
15 $\quad$ **forall** $\texttt{A}_i$ **do**
16 $\quad\quad SQ =$ sub-plan$(\sigma_{min_i \leq ?A_i \leq max_i}(\xi_{[\texttt{A}_i]}))$;
17 $\quad\quad$ add-synchronization-root(SQ,det-candidates);
18 $\quad\quad$ forward(init(SQ)) ; `/* routing depends on the chosen operators */`
19 $\quad$ **end**
20 **end**

Each physical skyline operator starts by determining the minima for all involved dimensions, see Algorithm 11. Separate query plans are created that represent the corresponding subqueries. After annotating these plans with information about synchronizing peer and method, the subqueries are initiated in line 4. The included operators should utilize the `forward(first-key())` method. The found values are inserted into the resulting plan copies and forwarded to materialize the missing dimensions for each found minimum. These two steps correspond to subqueries of the form $\omega_{[\texttt{A}_j]\forall j \neq i \wedge j < n}(\varphi_{MIN;?A_i;1}(\xi_{[\texttt{A}_i]}))$ for each attribute $\texttt{A}_i$. This could be done in sequence, but the three physical operators presented here query the minimum of each dimension in parallel. For ω each available implementation can be used. ω^{HASH} should be the preferred choice, because this results in only one message for each involved dimension. When all dimensions are materialized, the query plans are sent to the synchronization peer. At that peer, Algorithm 12 is executed. Synchronization in this phase is easy to achieve, because the number of dimensions is known and thus, the number of expected replies. At the synchronization peer, the resulting ranges for each attribute are determined. The minimal values provide the lower bounds, the maximal values over all materialized triple sets provide the upper bounds for each attribute. These ranges form the frame. From this point on, the three versions differ.

Using Φ^{SEQ} the synchronization peer initiates a range query for each dimension covering the ranges determined before. This is achieved using a combined operator $\sigma_{min_i \leq ?A_i \leq max_i}(\xi_{[\texttt{A}_i]})$ for each attribute $\texttt{A}_i$. The corresponding results have to be combined in the next synchronization phase, which is illustrated in Algorithm 13. For $\sigma^{SEQ}(\xi)$ synchronization is straightforward, because there will be exactly one reply for each attribute $\texttt{A}_i$. If $\sigma^{RQ}(\xi)$ is used, exact synchronization will be challenging, because each queried range will result in an unknown number of replies. One approach is to use constant waiting times. This leads to the difficult question of

Algorithm 13: `det-candidates()` for $\Phi_{\texttt{MIN},\ldots,\texttt{MIN};?A_1,\ldots,?A_n}(\omega_{?A_i;[\texttt{A}_1],\ldots,[\texttt{A}_{i-1}],[\texttt{A}_{i+1}],\ldots,[\texttt{A}_n]}(\xi_{[\texttt{A}_i]}))$

Input: attribute names $\texttt{A}_1 \ldots \texttt{A}_n$

1 $C = \emptyset$;
2 **forall** $\texttt{A}_i$ **do**
3 | $C_i = \emptyset$;
4 **end**
/* requires waiting times or completeness estimation */
5 **while** *ranges not completely covered* **do**
6 | $C_i = C_i \cup reply_{\texttt{A}_i}$; /* collect replies for the different dimensions */
7 | $C_{new} = \bigcap_{i \leq n} C_i \setminus C$;
8 | $C = C \cup C_{new}$;
9 | **if** *evolving* $\wedge C_{new} \neq \emptyset$ **then**
10 | | $S = BNL(C)$;
11 | | signalize(RES);
12 | **end**
13 **end**
14 **if** $n > 2$ **then**
15 | **if** *online* **then**
16 | | $S = BNL(C)$;
17 | | signalize(RES);
18 | **end**
/* determine projections and materialize only the missing attributes */
19 | $SQ = \text{sub-plan}(\omega_{[\texttt{A}_1],\ldots,[\texttt{A}_n]}(\{t | \exists C_i \exists C_j (i \neq j \wedge t \in C_i \wedge t \in C_j)\} \setminus C))$;
20 | add-synchronization-root(SQ,finish);
21 | forward(init(SQ)) ; /* routing depends on the chosen operators */
22 | forward(p_{sync},finish($\texttt{A}_1 \ldots \texttt{A}_n, C$));
23 **end**
24 **else**
25 | $S = BNL(C)$;
26 | signalize($DONE$);
27 **end**

how long to wait for replies. There is no reliable knowledge about how many peers are contacted and how many peers will reply to each range query. On the one hand, these times can be too high. In this case, the operator might still block further processing even if all results are received. On the other hand, the waiting times can be too low, which results in incomplete range query results. Further, late replies can block processing by far, but maybe result in no or only insignificant extensions of the final skyline. Thus, a more feasible approach is to estimate the completeness of range-query results and continue processing as soon as a defined completeness is achieved. How to achieve this is discussed in Chapter 7. In each case, the results for the ranges of attributes $\texttt{A}_i$ are collected in according sets C_i, as indicated in line 6. Points that are included in all C_i are located in the queried frame. Thus, they form the first skyline candidates C. `Evolving` and `online` are optional extensions described below.

In multidimensional (> 2) skyline queries also projections to two-dimensional subspaces of the search space may be part of the result. This is illustrated in Figure 6.7 for the three-dimensional skyline operator

$$\Phi_{\texttt{MAX},\texttt{MIN},\texttt{MIN};?hp,?price,?mil}(\omega_{?o;[\texttt{'price'}],[\texttt{'mil'}],[\texttt{'name'}]}(\xi_{[?o..?hp\rightarrow\texttt{'hp'}.]})).$$

The three extrema p_1, p_2, and p_3 define the frame, which forms a cuboid in this case. Each point that lies in this cuboid can be element of the skyline and will be included in the results of

p_1 : 140, 25, 60 (maximal hp)
p_2 : 70, 5, 80 (minimal price)
p_3 : 120, 30, 20 (minimal mileage)
p_4 : 60, 25, 30 (part of skyline)
p_5 : 110, 10, 100 (part of skyline)
p_6 : 130, 45, 40 (part of skyline)
p_7 : 110, 45, 40 (dominated by p_3, projection)
p_8 : 50, 35, 90 (dominated by p_2, no projection)
p_9 : 130, 35, 110 (dominated by p_1, no projection)

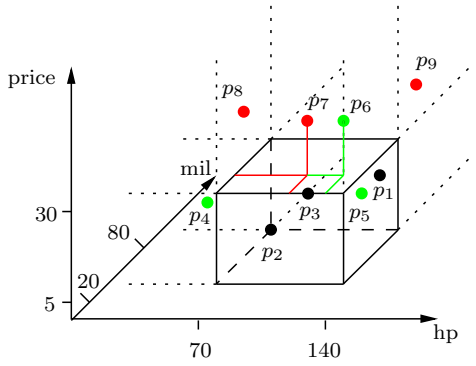


Figure 6.7: Projections in a three-dimensional skyline

all three queried ranges. The figure shows points that are not included in all three range-query results. p_8 and p_9 are no projections to two-dimensional subspaces and cannot be elements of the skyline. This is different for p_4 to p_7, which all are such projections. The figure shows that such points can be part of the skyline, but they may also be dominated by other points, e.g., p_7 is dominated by p_3. We indicate the actual projection for p_6 and p_7. In general, points from the frame can dominate projections, but cannot be dominated by projections. This is proven in [142]. Following, we extend the search space as indicated by the dotted lines. This is also proposed in [142] and [104]. In practice, we achieve this by materializing all missing attributes for all points that are included in at least two different ranges, but not in all ranges. This is implemented using an ω operator. Lines 14 to 23 of Algorithm 13 show how this extra materialization phase is initiated. For skyline queries over more than three dimensions an ω^{HASH} should be chosen. Otherwise, the missing dimensions would have to be materialized one after another. If only one missing attribute has to be materialized, also ω^{RQ} may be a suitable choice. Note that this is the only place where a peer initiates subqueries and sends an extra message to the synchronization peer. So, we do not have to ship all candidates with the initiated subqueries. In all other cases the subquery plans contain all information needed to further process the skyline query.

Actually, the search space could be pruned even more. For instance: we only have to check candidates from the extended search space with an `hp` value greater than that of p_3. This reveals p_7 as dominated by p_3 without extra materialization of dimension `price`. It has less horsepower and more mileage compared to p_3. If it would have a `price` value lower than that of p_3, it would have been included in the results of the range queries. Thus, p_7 could be omitted from the final materialization phase. We leave further discussions to future work. The irregularly formed resulting narrowed search space is indicated in Figure 6.8.

For two-dimensional skylines, a central BNL algorithm [33] is used to compute the global skyline (lines 25 to 26). If the extra materialization phase is needed, the central BNL algorithm will be executed at the final synchronization peer, see Algorithm 14. We call this a sequential implementation, because all phases are initiated by the synchronization peer(s) in sequence. Nevertheless, all range queries may be processed in sequence or in parallel. We use $\Phi^{SEQ}(\omega(\xi^{SEQ}))$ and $\Phi^{SEQ}(\omega(\xi^{RQ}))$ to distinguish between both. Similar options are available for the materialization phase in multidimensional skyline queries. We follow the same notation and write, for instance,

$$\Phi^{SEQ}_{\texttt{MAX,MIN,MIN};?hp,?price,?mil}(\omega^{HASH,PAR}_{?o;[\texttt{'price'}],[\texttt{'mil'}],[\texttt{'name'}]}(\xi^{RQ}_{[?o,?hp\rightarrow,\texttt{'hp'},]})),$$

which nevertheless refers to only one physical operator.

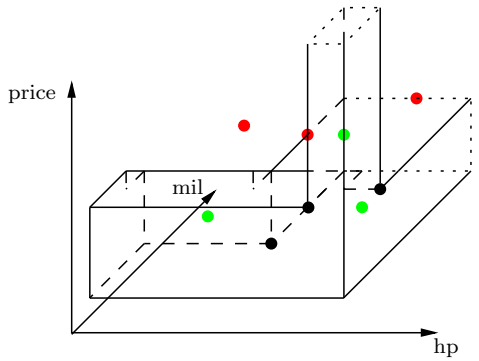


Figure 6.8: Narrowed search space in a three-dimensional skyline

Algorithm 14: `finish()` for $\Phi_{\texttt{MIN},\ldots,\texttt{MIN};?A_1,\ldots,?A_n}(\omega_{?A_i;[\texttt{A}_1],\ldots,[\texttt{A}_{i-1}],[\texttt{A}_{i+1}],\ldots,[\texttt{A}_n]}(\xi_{[\texttt{A}_i]}))$

Input: attribute names $\texttt{A}_1 \ldots \texttt{A}_n$, candidates C

```
1 while not all candidates materialized do
2 |   C = C ∪ reply;
3 |   if evolving then
4 |   |   S = BNL(C);
5 |   |   signalize(RES);
6 |   end
7 end
8 S = BNL(C);
9 signalize(DONE);
```

Though Φ^{SEQ} is implemented stateless, the different phases show facets of the data shipping approach. All data are shipped to the synchronization peer(s) and processed there. Φ^{PAR} and Φ^{HYB} try to overcome problems resulting from this. After determining the frame borders, not all of the resulting ranges are queried directly. Rather, separate subskyline queries are sent to all peers responsible for the range of one dimension. Which dimension is chosen can significantly influence performance. We suggest to use the smallest range, because in most cases this will correspond to the fewest peers. This is shown in lines 11 to 12 of Algorithm 12. This can be improved by using knowledge about the structure of the DHT. In analogy to the sequential variant, each contacted peer uses range queries and materialize functionality to collect only the remaining attributes and computes local skylines. If Φ^{PAR} is used, a separate materialization phase is initiated at each peer that received a subskyline query. The resulting local skylines are sent to the synchronization peer, which computes the final global skyline. This variant comes along with higher communication efforts. But, due to the distributed processing, it provides better query answer times. Figure 6.9 illustrates the processing of Φ^{PAR}. First, the indicated frame is determined by looking up the minima for all dimensions and afterwards materializing the missing attributes. Then, four query plans are sent independently to the peers responsible for attribute x. Each of them computes a local skyline. Points of the local skylines may be deleted from the final global skyline, because they are dominated by values stored at other peers.

Φ^{HYB} first follows the same approach as Φ^{PAR}. But, with Φ^{PAR} the peers that determine local skylines have to materialize candidates independently in the final step. Thus, several candidates will be materialized multiple times by different peers. This is because each of these peers queries the same ranges and thus, receives the same results. Projections that are

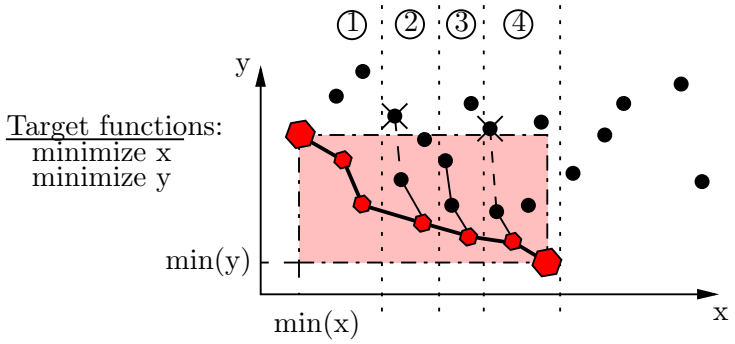


Figure 6.9: Principle of Φ^{PAR}

independent from the dimension chosen for subskyline computing are handled by each peer. To avoid this while still utilizing the power of distributed computing, only the final materialization step is delegated to a synchronization peer again. There, duplicate candidates are detected and materialized uniquely. Based on the local skylines and the materialized candidates, the final skyline is computed. This reduces the communication overhead at costs of answer times.

An important challenge in loosely coupled and unreliable systems is to avoid waiting states. The introduced operators follow the stateless fire-and-forget approach, but they cannot be implemented without synchronization phases. This is unavoidable for ranking operators. Even if we enhance the operators by applying completeness estimation as indicated above, it may be desired to provide first results as soon as they are available. Thus, we propose two extensions of each of the three variants: *online* and *evolving* processing. With the online extension, operators provide parts of the skyline as soon as they are certain to belong to the final skyline. This first skyline can be determined after receiving all range query answers, because points from the frame cannot be dominated by any projection. Thus, the online version has effect only for skyline queries with more than two dimensions. In the evolving version even elements are provided that could finally be dominated by elements determined later – the skyline may evolve. This can be done after receiving any range query answer. All points for which all dimensions are already materialized can be used as input for an according temporary skyline. This can even be extended to start materializing candidates as soon as first range query results are received. But, this involves overhead, because the attribute values that are materialized may also arrive with later range query replies. In Algorithm 13 we omit this second option. Both extensions, online and evolving, are tailor-made for dynamic and unreliable environments. Of course, this requires some mechanisms to process temporary results before finally finishing the skyline operator. This is introduced as *pipelining* in Section 6.4.

6.3 Other Operators

Several other physical operators are not discussed in detail. On the one hand, this is because they rely on the same principles as the operators already introduced. For instance, we mentioned combined operators like $\sigma^{RQ}_{x\leq?A\leq y}(\xi_{['\mathtt{A}']})$, which utilizes range queries as $\xi^{RQ}_{['\mathtt{A}']}$ does. Further, the q-gram-based similarity operators can be extended to implement predicates like `starts-with()` (prefix relation), `ends-with()` (suffix relation), and `contains()` (substring relation). For this, the according amount of q-grams has to be extracted from the search string.

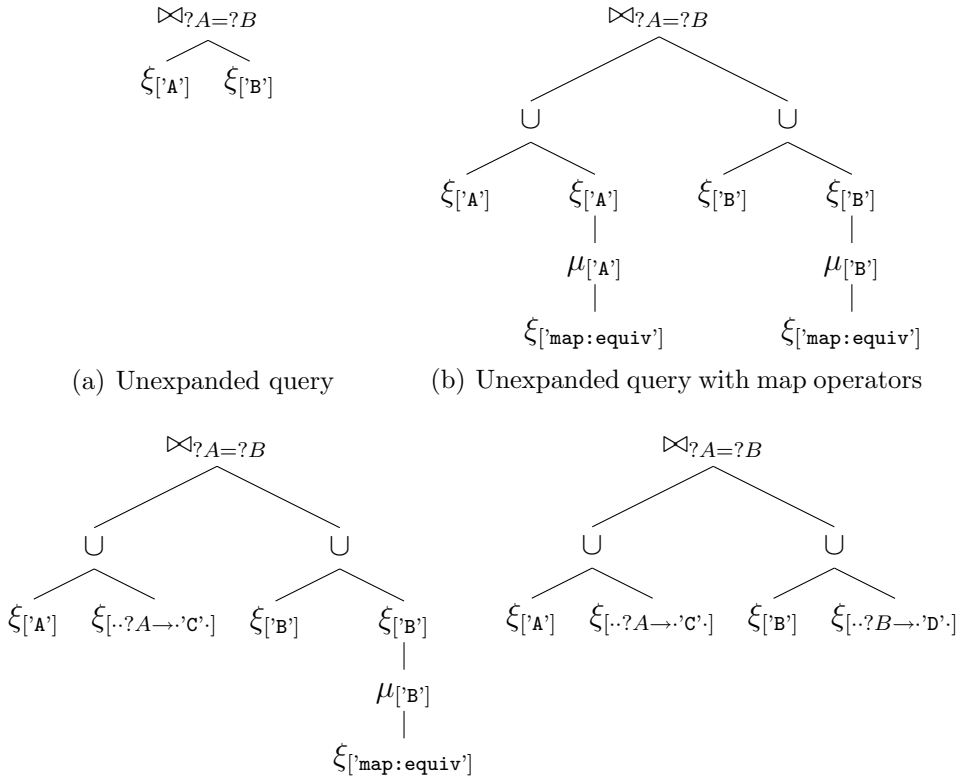
Then, the processing is analog to the processing of similarity queries. The efficiency of this is leveraged if we use positional q-grams.

On the other hand, several operators are implemented straightforward and do not need any further discussions. One such operator is the union operator $\cup$. There is no sophisticated approach for implementing its semantic, because it just refers to combining the input triple sets. The only specific point is the possibility of processing its branches in parallel. This is discussed in the context of processing query plans in Section 6.4.

In Section 5.1 we also mentioned the grouping operator γ. Up to this point, we integrated a very simple version of this operator into the proposed framework. It is capable of determining simple aggregates like the average on a set of input values. Its intended use was to show the impact of estimating query completeness, which is introduced in Section 7.2. Thus, it is based on a rather simplistic algorithm that adheres to the principle of synchronization. Following, there are two physical variants available: γ^{LOC} can be used in query plans that contain only sequential operators and γ^{SYNC} is designed for query plans with parallel operators. For large-scale distributed systems, it is advisable to distribute synchronization load across multiple peers. The most popular way of doing so is to build aggregation trees during query time. In this approach, the different synchronization peers are arranged in a hierarchy and determine subaggregates. The synchronization peer at the root of the tree produces the final aggregate value(s). [44, 99] provide example implementations. An alternative to γ^{SYNC} is to accordingly utilize existing indexes. For instance, the attribute-value index clusters data of groups at single peers. If this is accompanied by an ω operator that respects the group semantic, all additional values can be materialized and aggregated without the need for explicit synchronization. Another approach is to collect all needed values using existing extraction and materialization operators. For final grouping without the need for explicit synchronization, all resulting query plans can be sent to peers responsible for the grouping values. This also uses the fact that data are clustered accordingly at these peers. All indicated variants can enable online processing when combined with pipelining. We leave further investigations to future work and refer to the wealth of existing works on aggregate queries in widely distributed systems, e.g., [44, 99].

A last operator that should be briefly discussed is the map operator μ. This operator is not a routing operator. Rather, it depends on preceding routing operators used for extracting and materializing the needed semantic metadata. Thus, we propose only μ^{LOC}. The efficiency of integrating semantic correspondences into query processing depends on how the additional semantic information is queried. The specific requirements are: *(i)* to prevent a negative effect of query expansion on the performance of the original, unexpanded query. In the first line, this means to prevent increasing response times for results belonging to this original query. *(ii)* load injected into the system caused by additionally querying mapping data has to be minimized. This mainly concerns the number of additional messages generated and the amount of additional bandwidth used. For semantic expansion, corresponding parts of a query plan are cloned and added as new children of a map operator, which precedes a union operator. An according subquery beneath the map operator is used to resolve correspondences. The more complex the subquery is, the more peers will be involved. We briefly illustrate this using a small example query:

```
expand A 1
expand B 1
select ?A,?B
where { ?x A ?A . ?y B ?B filter ?A=?B}
```



(a) Unexpanded query (b) Unexpanded query with map operators

(c) First resolved mapping (d) Completely expanded query

Figure 6.10: Example query expansion

Figure 6.10 shows the corresponding query plan and the different steps of query expansion, assuming that `A` corresponds to an attribute `C` and that `B` corresponds to an attribute `D`. First, the original unexpanded query is extended by map operators. To achieve the first requirement introduced above, to not effect the response times, both sides of a union operator should not be processed in post-order. Post-order processing would result in the following sequence:

1. Attribute `A` is extracted.
2. Correspondences for `A` are resolved and the query is rewritten as shown in Figure 6.10(c).
3. Attribute `C` is extracted.
4. Attribute `B` is extracted.
5. Correspondences for `B` are resolved and the query is rewritten as shown in Figure 6.10(d).
6. Attribute `D` is extracted.
7. The join is processed.
8. Original and additional result data are returned in conjunction.

This obviously delays the delivery of original results. Thus, the branches of a union operator should be processed in parallel. We call this *inter-operator parallel* processing and discuss it

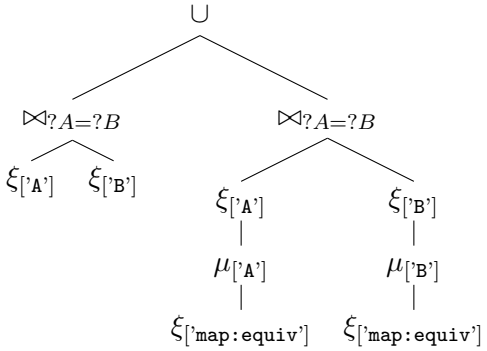
in Section 6.4. As the actual union of input sets is data independent, it can be processed on arbitrary intermediate nodes during query processing. The best way is to combine results at the query initiator. Inter-operator parallelism would result in extracting attribute `A` and in parallel resolving correspondences for `A`. For resolving correspondences, according copies of the query plan are sent to peers responsible for attribute `map:equiv`. On these peers, the mapping operators are finished, the dependent routing operators are modified to reflect the mappings, and the query can be processed further. The ξ operator used to extract `map:equiv` may be a parallel variant. If several peers are responsible for the attribute, each of them can independently rewrite the query and continue processing. For more complex subqueries, this might require synchronization before rewriting. If multiple attributes that correspond to `A` exist, this will result in multiple branches of the union operator. While the correspondences are resolved, query plans containing triples belonging to `A` are already forwarded to process the right branch of the join. The second union operator is also processed inter-operator parallel. Thus, while resolving the correspondences for `B`, triples belonging to `B` are already extracted, joined, and returned. Using this parallelized approach, result data from the original query is materialized as fast and efficient as before. Semantic expansion results in expanded query results, which usually will result in extended processing time for the whole query. We will further argue on this in Section 8.2.6. The advantages of the parallel approach are:

- processing is still highly stateless and thus, scalable
- efficiency is provided due to the high grade of parallelization
- automatic expansion is cost-based and solely due to the query engine

The generic nature of the approach, based on functionality implemented in existing operators, allows for arbitrarily combining different triple parts. This flexibly supports different representations of semantic data. The exact overhead will depend on the kind and complexity of extracted semantic data.

The example highlights several further issues. First, an inter-operator parallel variant of the join operator should be provided as well. Then, all branches would be processed in parallel. Besides accelerating the resolving of mappings, this could even decrease response times for original results. Thus, this physical operator should be considered in general. Note that an inter-operator parallel join requires explicit synchronization. Another issue is the need of identifying and reusing shared expressions. In the example, `map:equiv` should be extracted only once and all mappings should be processed in one step. This has to be considered accordingly when deciding on inter-operator parallel processing.

A last issue is the flexibility of where to place the union operator. For instance, another option for the unexpanded query with mapping operators is:



This results in a semantic different to that from Figure 6.10. Only values for `A` and `B` as well as `C` and `D` are joined. There will be no join between values for `A` and `D` or `C` and `B`. In cases,

this may be a desired behavior. We do not discuss these issues in more detail. The aim of this work is to provide flexible functionality to tightly integrate semantic correspondences into query processing, but not to handle the full depth of semantic correlations. For the process of navigating through the mapping graph we refer to [58] and similar works. More details on the implementation of the map operator, including issues of transitivity and complexity of rewritten queries, can be found in [138].

6.4 Processing Query Plans

Up to here, we discussed logical and physical query plan operators. During this, we introduced sequential processing and intra-operator parallel processing. We also indicated how complex query plans are processed and briefly introduced inter-operator parallelism. In this section, we get back to this and discuss strategies, alternatives, and concepts of query plan processing in more detail. The crucial questions handled here are concerning the order in which operators are processed, at which time, and by which peers.

Query plans have to be processed from bottom to top. This restricts the alternatives of processing query plans – and highlights the importance of the operator order resulting from the process of query plan optimization. The basic strategy is to process plans in post-order. Certain operators may be combined to enable a more efficient processing. This should be considered in the process of logical optimization. Since the impact of combining operators can only be evaluated meaningfully on the basis of cost estimations, a reordering of operators should also be enabled during the process of physical optimization. We briefly discuss query planning below in Section 6.4.3.

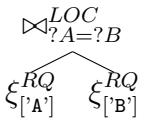
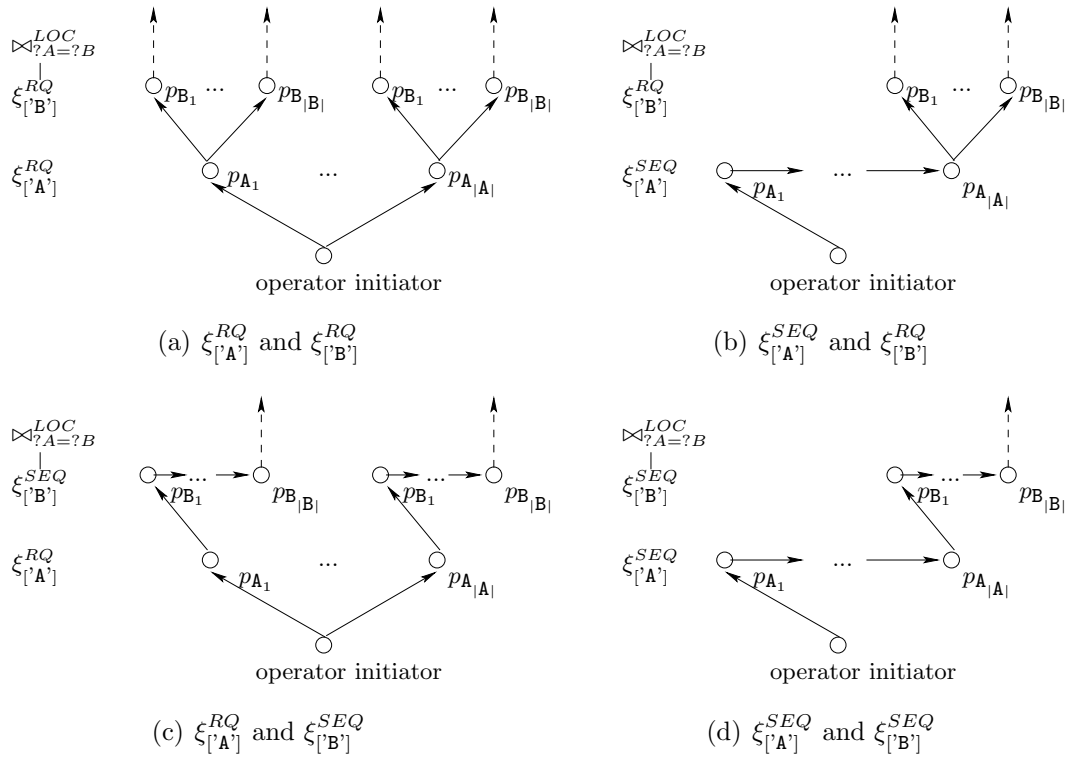


Figure 6.11: Example query plan

Query plan processing is based on the idea of *Mutant Query Plans* (MQP) [163]. We present some details of this work in Section 6.6. Here, we illustrate it using a small example query plan as shown in Figure 6.11. The query plan is shipped to one or multiple peers responsible for the next operator to process. These peers insert all data that correspond to the operator. In the example, the plan is first shipped to all peers responsible for attribute `A`, where all local triples t with $t[a]$ = `A` are inserted. Adhering to the post-order processing, the query plans are next shipped to the peer(s) responsible for attribute `B`. Again, local triples are inserted. Like this, the plan mutates, because its operators are successively replaced by according data. If $\xi^{SEQ}_{['A']}$ is used, one plan containing all data for `A` will be sent. In contrast, if $\xi^{RQ}_{['A']}$ is used, each peer responsible for a part of `A`'s data will be contacted in parallel. Thus, multiple plans, each containing a part of the data, are forwarded. This is an extension of the original MQP concept. During the processing of each operator, peers can autonomously duplicate plans as well as change its structure and the data contained. By this, we add even more possibilities of mutations. Thus, we call this concept *Mutating Mutant Query Plans* (M^2QP). This allows a higher degree of parallelism, because all plans are traveling through the network independently. But, this involves more messages, because each plan that contains a part of `A`'s data has to be sent to each peer responsible for a part of `B`'s data. Otherwise, we would probably miss matching pairs. As soon as the data of one operator is not needed anymore it is deleted from



(a) $\xi^{RQ}_{['A']}$ and $\xi^{RQ}_{['B']}$ (b) $\xi^{SEQ}_{['A']}$ and $\xi^{RQ}_{['B']}$

(c) $\xi^{RQ}_{['A']}$ and $\xi^{SEQ}_{['B']}$ (d) $\xi^{SEQ}_{['A']}$ and $\xi^{SEQ}_{['B']}$

Figure 6.12: Different combinations of ξ operators ($|A| := |p_{r_{\mathtt{A}}}|, |B| := |p_{r_{\mathtt{B}}}|$)

the plan. In the example, each peer responsible for attribute B can process a part of the join. Thus, only matching pairs are inserted into the plan, replacing the $\bowtie$ operator. Afterwards, the output data of the ξ operators are not needed anymore and can be removed. If $\xi^{SEQ}_{['B']}$ is used, processing of $\bowtie$ and deletion of input data will happen at the last peer in the sequence of peers responsible for B. If $|p_{r_{\mathtt{A}}}|$ refers to the number of peers responsible for the range of A and $|p_{r_{\mathtt{B}}}|$ refers to the number of peers responsible for B, the different combinations of physical operators will result in the following number of final query plans, i.e., replies. Figure 6.12 shows this using the same graph-based illustration as used when introducing the physical operators in Section 6.1. We call these illustrations *routing graphs*.

- $\xi^{RQ}_{['A']}$ and $\xi^{RQ}_{['B']}$ (Figure 6.12(a)): $|p_{r_{\mathtt{A}}}| \cdot |p_{r_{\mathtt{B}}}|$ query plans
- $\xi^{SEQ}_{['A']}$ and $\xi^{RQ}_{['B']}$ (Figure 6.12(b)): $|p_{r_{\mathtt{B}}}|$ query plans
- $\xi^{RQ}_{['A']}$ and $\xi^{SEQ}_{['B']}$ (Figure 6.12(c)): $|p_{r_{\mathtt{A}}}|$ query plans
- $\xi^{SEQ}_{['A']}$ and $\xi^{SEQ}_{['B']}$ (Figure 6.12(d)): 1 query plan

A crucial point is the question of how to handle query plans with empty output data of operators above the leaf level. In order to save on the number of generated messages and used bandwidth, they should be "dropped", i.e., they should not be forwarded. This is possible after an operator signalizes *DONE* and is replaced by an empty result set. But, this is not applicable for $\cup$ operators. This is a strong reason to process the branches of such data-independent operators in parallel using inter-operator parallelism. Inter-operator parallelism is an alternative to the

post-order processing of query plans. Using this strategy, the children of a binary operator are processed in parallel. To process the binary operator, resulting query plans have to be collected at a synchronization peer. Inter-operator parallelism is especially useful for operators like the union operator. This operator does not require explicit synchronization during query time. Since the synchronization is data-independent, it degrades to a simple combination of result sets. Other operators that can benefit from inter-operator parallelism are join operators $\bowtie$ and cross products $\times$. Here, the two child branches can be processed in parallel, but need explicit synchronization before processing subsequent operators. In this context, we have to point out that parallelism implies redundancy. For instance, in the example from Figure 6.11, each peer responsible for a part of `B` will receive and process as many query plans as peers are responsible for `A`. In the context of a join this is fine, because each input plan contains different data and only matching pairs are forwarded. But, in the context of a data-independent binary operator like $\cup$, this implies unnecessary overhead. Thus, inter-operator parallelism should be applied for such operators in general.

Synchronization itself offers some options. Intuitively, all input data should be collected and afterwards one final result of the operator should be produced. We introduced this as blocking processing. This may delay query times too much and reveals the challenge of identifying the point in time when all intermediate results are received. The idea of online processing based on multiple result revisions may also be applied to inter-operator parallelism. For ranking operators, the result elements and their order may change in each new revision. For operators like joins online processing extends the result set iteratively. This provides a convenient way of producing fast results. In many situations users may even be satisfied with such a rough – but fast – overview. As mentioned before, this can only be applied in conjunction with some pipelining mechanism. This is discussed next.

6.4.1 Pipelining

Algorithm 15: ω_P and $\xi^{SEQ}_{[\texttt{'A'}]}$ combined by triple set pipelining

```
   Input: attribute name A, predicate P
   /* ...                                                    */
 2 forall t ∈ OT_local(t[a] = A) do
21   if evaluate(P, t) = true then
 3     add-to-result({t});
22   end
 4 end
   /* ...                                                    */
```

Pipelining is a technique known from traditional DBMS, in which it is used to speed up query processing. As soon as output of an operator is available, it is passed to the next operator in the query plan and processed further. Like this, several operators can be processed in parallel and the processing of subsequent operators is delayed only to a minimum. We refer to this as *triple set pipelining*. The processing of extract and materialize operators discussed in Section 6.1 provides an example. If a selection σ_P succeeds these operators, extracted or materialized triples may be directly passed to this operator. This corresponds to inserting an according line in the operator algorithms. For instance, in Algorithm 1 on page 92, the two new lines 21 and 22 are inserted between lines 2 and 4 as shown in Algorithm 15. This makes storing the output triples of ξ unnecessary. Rather, the σ operator is directly replaced by the according set of output triples. Obviously, triple set pipelining is useful to speed up local processing of

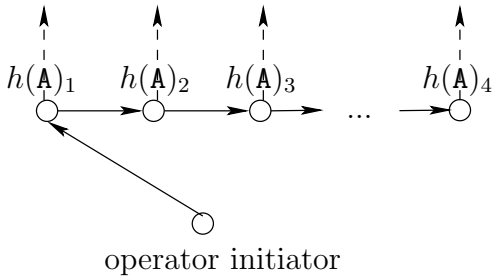


Figure 6.13: Routing scheme for $\xi^{SEQ}_{['A']}$ with peer pipelining

query plans. If it is not used, we will first finish the extract operator and afterwards process the selection. Triple set pipelining can also influence bandwidth costs. For instance, if an operator combination $\sigma^{LOC}_{P}(\xi^{SEQ}_{['A']})$ is processed without pipelining, the query plan will first pass all peers responsible for attribute A. Each peer inserts local data before forwarding the plan. σ^{LOC}_{P} will be processed at the last peer in the sequence, filtering out all data that does not match P. A more efficient approach is to filter data locally at those peers that insert them into the query plan, which is enabled by triple set pipelining. Triple set pipelining has no impact on the number of generated messages. As a modification, according operators can be processed in parallel. This is called pipelined parallelism [217] in traditional DBMS.

We introduce a modified version of pipelining called *peer pipelining*, which is specifically designed for widely distributed systems. The concept is basically the same, but does not only involve operators that are processed at the same peer. Rather, it is extended to include also routing operators. This is mainly effective in sequential operators, but also in synchronizing operators like Φ and inter-operator parallel joins $\bowtie$. The idea is to pass intermediate results to routing operators like ω as soon as local processing or an operator phase is finished. Figure 6.13 illustrates this for ξ^{SEQ}. In the usual case, we first pass the whole sequence of peers responsible for the extracted attribute. Afterwards, one query plan containing the whole set of output triples is forwarded to the peer(s) responsible for processing the next operator. If one peer in the sequence fails or is slow, this may break or delay the chain of processing unnecessarily. With peer pipelining, the output produced so far is forwarded immediately. This is still different to parallel processing, i.e., to choosing $\xi^{RQ}_{['A']}$. Only the first peer of the sequence is contacted by the initiating peer. The approach results in the same number of forwarded query plans as in the parallel case. But, the fact that all peers are contacted in sequence eases the estimation of the number of resulting query plans. Further, sending multiple independent query plans to the same peer is prevented, which can be guaranteed only in sequential operators. This still comes with an increased vulnerability to broken chains and slow peers. Peers succeeding a slow peer in a sequence are still delayed in processing. Since peer pipelining increases the number of generated messages, it should not be applied per produced output triple set. Rather, it should be applied per peer that finishes a part of processing (sequential operators) or per revision (synchronizing operators). Peer pipelining is a prerequisite for enabling online and evolving processing. In this context, a final projection π has to be understood as a routing operator. Each time a new revision is produced, the variable values can be projected and sent to the query initiator. Only then online and evolving processing can improve search experience of the user.

6.4.2 Overview

Algorithm 16: General plan processing `process(`Q`)`

```
   Input: query plan Q
 1 op = get-next-operator(Q);
 2 if is-logical(op) then
 3 |   op = plan-query(Q);
 4 end
 5 process-operator(op);
 6 if is-done(op) then
 7 |   prune-plan(Q);
 8 |   process(Q) ;                          /* restart with next operator */
 9 end
10 else
11 |   if peer pipelining ∧ new-data-available(op) then
12 |   |   C = clone-plan(Q);
13 |   |   forward(C);
14 |   |   mark-partly-processed(op);
15 |   |   mark-done(op);
16 |   |   process(Q) ;                      /* restart with next operator */
17 |   end
18 |   else
19 |   |   forward(Q);
20 |   end
21 end
```

An overview of the general process of query plan processing is provided in Algorithm 16. We illustrate only the main steps. This procedure is executed at each peer that receives a query plan copy. First, the next operator to process is determined. This respects the options of post-order processing and inter-operator parallelism. If that operator is still a logical operator (at query start or during query time when dynamic planning is enabled – see Section 6.4.3), the process of query planning is initiated. Here, one or multiple logical operators are replaced by physical ones, synchronization peers are chosen, and operators may be reordered. Afterwards, `process-operator()` is called for the current operator. If no data for processing is available, the operator will return without doing anything. Otherwise, all data (including local data) that can be processed are processed. Result data are inserted into the query plan Q. If triple set pipelining is enabled, this will be integrated here as well. If the operator finishes, the $DONE$ flag will be set. This flag decides whether the `process()` method is restarted or not. If it is restarted, this will result in processing the next operator. Otherwise, the query plan Q is pruned (remove unnecessary data and operators) and forwarded to the peer(s) that are the next to process the current operator. This forwarding implements the different routing calls introduced for the physical operators. Also, finally processed queries are returned to the query initiator and inter-operator parallelism is initiated in the `forward()` method. The integration of peer pipelining is illustrated in lines 11 to 17. The created plan copy C is forwarded to continue the processing of the current operator. Both query plan copies can be pruned in this step. The current operator in the local plan copy Q is marked as partially processed and $DONE$, which results in processing the next operator by restarting the plan processing procedure. If the next operator is a routing operator, this results in forwarding the original plan copy as well. Following this procedure at each peer, each created plan copy will be replied at some point in time. Next, we discuss the process of query planning.

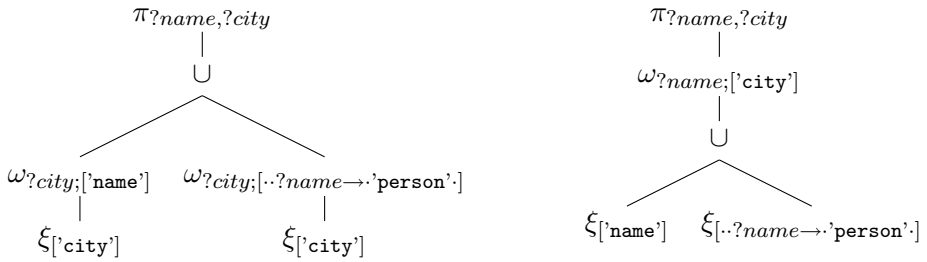
6.4.3 Query Planning and Adaptive Query Processing

Query planning is the process of transforming a VQL query into a physical query plan. This involves logical and physical plan optimization. In Section 5.1 we introduced some basic rules for transforming query constructs into (parts of) a logical query plan. This builds a basis for the logical optimization. Logical optimization involves simplification of expressions, union or splitting of operators, and other rule-based optimizations. For this, methods from traditional DBMS can be adopted. General optimization rules are similar to those that are known from relational DBMS, such as pushing selections σ downwards and pushing materializations ω upwards. Further issues that should be regarded have been indicated throughout this work and are not discussed here in detail. [190] provides some more insights into the process of logical query optimization.

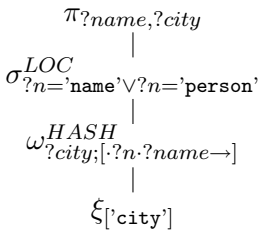
To get an executable plan, the contained logical operators have to be replaced by physical operators. This is matter of the physical optimization. Using static planning, this happens at the peer initiating the query. The optimizer has to be aware the fact that there exists an $m : n$ mapping between logical and physical operators. This means, there exist several physical operators for one logical operator. Further, multiple logical operators can be combined in one physical operator and vice versa. Physical optimization should also support the reordering and combination of operators. To meaningfully enable this, some tasks that could be integrated into logical optimization should be delayed after the physical one. For instance, this regards the combination of selection operators with atomic predicates. This is because for supporting combined operators like $\sigma^{QG}(\xi)$ the optimizer has to identify atomic predicates. Further, several operator orders can only be identified and evaluated with the knowledge about available physical operators. The following VQL query is an intuitive example:

```
select ?name, ?city
where { ?o 'name' ?name . ?o 'city' ?city
        union
        ?p 'person' ?name . ?p 'city' ?city }
```

Two equivalent logical query plans that could be evaluated in the process of logical optimization are:



For processing the first plan the shared expression $\xi_{['city']}$ should be identified and processed accordingly. The second variant utilizes the fact that attribute `city` exists for both input sets of the union. The logical optimization cannot consider the following third alternative, which utilizes the possible combination of σ^{LOC} and ω^{HASH}. Assuming the support of disjunctions in σ, the union can be modeled by an `OR` and processed without intermediate routing, which is not possible with operators like ω^{RQ}:



A cost model should be applied to choose the best combination of physical operators. Implementing sophisticated optimizers is a complex and challenging task. In Section 6.5 we provide a basis for this by discussing costs for physical operators presented in this work. But, due to limited local knowledge, changing network situations, dynamics of load and peers, etc., static planning will usually not result in the best suited query plans. Rather, a dynamic planning should be preferred.

Technically, achieving dynamic planning is rather simple. The idea is to not replace all logical operators by physical ones before starting the processing of a query plan. Rather, only the operator(s) on leaf level have to be chosen in order to start query processing. Each time an operator is finished and the next operator has to be processed, we can check if this is still a logical operator. If so, this is the latest point we have to replace it by a physical implementation. Before that, the operator can be replaced at any time of query processing and the choice of operators can be restricted a priori. [174] achieves this by shipping so-called plan rewriters along with a query plan. Dynamic planning should make use also of cost estimations for the different physical operators. Like this, it represents a first approach of enabling adaptive query processing.

Adaptive techniques should be applied on more levels than only choosing physical operators. It is a main ingredient of the proposed M^2QP approach. By allowing each peer to autonomously change the structure of a query plan, we provide a maximum of flexibility in query planning and processing. It can be used for supporting certain advanced operators on the basis of existing ones. We illustrate this using the example of a q-gram-based join in Figure 6.14.

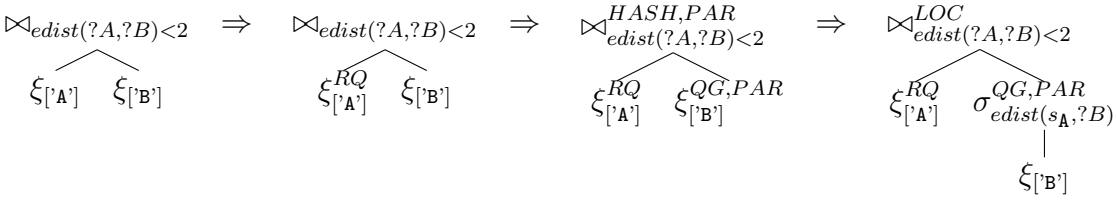


Figure 6.14: Example of utilizing mutations for adaptive query processing

Assuming a simple similarity join on the values of two attributes `A` and `B`, we start with a query plan as shown in the left of the figure. At the query initiator, we start query planning by replacing only the first extraction operator, as shown in the second query plan. Each peer responsible for a part of attribute `A` receives a copy of that plan, inserts local triples, and continues with processing the next operator. On this point, the peers have to decide which physical operator to choose for $\xi_{['\texttt{B}']}$. Depending on local knowledge and estimated costs, they may decide to use a parallel q-gram-based similarity join that looks up the q-grams of each input string $s_{\texttt{A}}$ from attribute `A` in parallel. The final plan looks as the third plan in the figure. To finally process the plan while utilizing existing operator implementations, the peers can generate one plan for each input string $s_{\texttt{A}}$ as shown in the end of the sequence. This is only possible if they can manipulate query plans autonomously. Note that triple set pipelining

should be applied to prevent the doubled evaluation of *edist*(). Similar insertions of temporary operators are used, for instance, to process the different phases of a skyline operator.

A powerful aspect of M^2QP is that each peer can decide autonomously. Thus, some may decide to use an operator implementation as in the example. Others may decide to query the q-grams of each input string in sequence, using $\bowtie^{HASH,PAR}_{edist(?A,?B)<2}(\xi^{QG,SEQ}_{['B']})$. Some can even decide to use a local join and process $\bowtie^{LOC}_{edist(?A,?B)<2}(\xi^{RQ}_{['B']})$. As all plans travel the network independently, they do not influence each other and the final result will be correct in each case.

(a) Late synchronization, more messages, increased robustness

(b) Early synchronization, less messages, decreased robustness

Figure 6.15: Late vs. early synchronization

Adaptivity in query processing has its limits as well. For instance, if a query plan contains a ranking operator φ, the peers cannot decide absolutely autonomously about he chosen implementation. If any of the operators preceding φ is executed in parallel, φ^{LOC} cannot be chosen. Rather, all intermediate query plans have to be routed to the *same* synchronization peer. That peer has to be defined before starting parallel processing. This applies to all operators that need some synchronization. A good choice is to use the query initiator as the synchronization peer and shift according operators in the plans upwards. Then, distributing query load through parallelization can help to increase robustness, because ranking is processed locally on all received "final" replies. In contrast, shifting ranking operators downwards can reduce the number of generated messages – on the sake of reduced robustness and increased query answer times. Figure 6.15 exemplarily illustrates this. Note that the join has to be processed before the ranking can happen. Otherwise, results could be inaccurate. Thus, the special implementations φ^{SEQ} and φ^{PAR} cannot be chosen. Further discussions on adaptive query processing techniques are an interesting issue for future work.

6.5 Costs of Physical Operators

Query planning involves the process of physical plan optimization, which is based on cost estimations. In this section, we provide a fundamental basis for a cost-based optimizer by providing cost formulas for the operators presented in this work. Further, this section is meant to summarize and compare the physical operators. As network access is still the most expensive resource, the main objective of the physical optimization is to minimize bandwidth and reduce communication latency. This reflects to minimizing the number (and size) of generated messages. Further, achieving good query answer times and robustness in dynamic environments is modeled by the number of hops needed to process a query plan. The difference between the number of messages and the number of hops expresses the degree of parallelism. If three

messages are sent in parallel, this will result in a hop count of only one. For complex plans, the number of total messages is determined by addition, the number of hops by applying the maximum function. Thus, if each from the three peers contacted above forwards a message to three more peers, this will result in $3 + 3 \cdot 3 = 12$ total messages and $1 + 1 = 2$ hops. If one of these peers initializes a sequential forwarding to contact its three peers, this will still result in $3 + 2 \cdot 3 + (1 + 1 + 1) = 12$ messages, but $1 + \max(1, 1 + 1 + 1) = 4$ hops.

symbol	explanation
N	number of key space partitions, i.e., unique P-Grid paths
D	total size of data managed in the system
m_{op}	messages generated for processing operator *op*
h_{op}	hops needed for processing operator *op*
f_{op}	number of follow-up query plans generated by operator *op*
m_l	messages for looking up one hash key
h_l	hops for looking up one hash key
$r_{\texttt{A}}$	range of an attribute `A`
$\lvert p_r \rvert$	number of peers responsible for range r
f_{in}	follow-up query plans of preceding operator, i.e., the number of input query plans
$\lvert T_{in} \rvert$	number of input triple sets combined over all f_{in} input query plans, $\lvert T_{in} \rvert \geq f_{in}$ (no empty input plans)
$\lvert p^x_{T_{in}} \rvert$	number of peers responsible for input triple sets T_{in} with respect to $x, x \in \{oid, a, v\}$
d	queried distance
$\lceil [\ldots] \rceil$	lower-than relation (upper limit)
$\lfloor [\ldots] \rfloor$	larger-than relation (lower limit)
rev_{op}	rate of producing revisions for a synchronizing operator *op*, $0 \leq rev_{op} \leq 1$

Table 6.1: Symbols used in cost formulas

Exact query answer times are impossible to predict. They could only be roughly estimated by assuming average times for message passing, waiting, processing, etc. These times would have to be summed up using the average number of hops a query needs. Thus, the number of query hops usually reflects query answer times very well. Consequently, for each operator *op* we focus on three cost factors:

1. The number of generated messages m_{op}.
2. The number of hops h_{op}.
3. The number of resulting follow-up query plans f_{op}.

The first two factors are sufficient for evaluating the costs of single operators and deciding on the chosen implementation. But, a high degree of parallelism results in a high number of follow-up plans, which cannot be respected in that context. If several parallel operators are processed in sequence, this can result in a message explosion. To respect this issue in a cost-based optimizer, we also have to take the number of messages generated by the processing of a whole plan into consideration. For this, we additionally use the third cost factor.

The cost formulas are based on estimating costs for processing a single lookup query in the DHT. Usually, for m and h these costs are guaranteed to be logarithmic in the number of peers. We use m_l to denote the average number of messages a lookup results in and h_l for the average number of hops. The symbols used in the formulas are summarized in Table 6.1.

To provide a rough overview, we list asymptotic limits for m and h for all routing operators in Table 6.2. They highlight the general differences between the operators. For instance, parallel hash-based operators depend on the total data size D (one query plan for each input triple set) and the number of participating peers (lookup costs for single query plans), reflected by the number of key space partitions N. In contrast, range-based operators depend on only N. Intuitively, the asymptotic costs are not suited to implement a cost-based optimizer. First, the

operator op	m_{op}	h_{op}
$\xi^{SEQ}, \omega^{SEQ}, \sigma^{SEQ}$	$O(\log N)$	$O(N)$
$\xi^{RQ}, \omega^{RQ}, \sigma^{RQ}$	$O(\log N)$	$O(\log N)$
$\omega^{HASH,SEQ}$	$O(D \cdot \log N)$	$O(D \cdot \log N)$
$\bowtie^{HASH,SEQ}$	$O(D \cdot \log N)$	$O(D \cdot \log N)$
$\omega^{HASH,PAR}, \bowtie^{HASH,PAR}$	$O(D \cdot \log N)$	$O(\log N)$
$\sigma^{QG,SEQ}$	$O(\log N)$	$O(\log N)$
$\sigma^{QG,PAR}$	$O(\log N)$	$O(\log N)$
$\bowtie^{HASH,SEQ}$ $(\xi^{QG,SEQ})$	$O(D \cdot \log N)$	$O(D \cdot \log N)$
$\bowtie^{HASH,PAR}$ $(\xi^{QG,SEQ})$	$O(D \cdot \log N)$	$O(\log N)$
$\bowtie^{HASH,SEQ}$ $(\xi^{QG,PAR})$	$O(D \cdot \log N)$	$O(\log N)$
$\bowtie^{HASH,PAR}$ $(\xi^{QG,PAR})$	$O(D \cdot \log N)$	$O(\log N)$

Table 6.2: Asymptotical cost approximations

costs of complex plans cannot be estimated. Information mandatory for that (e.g., the number of follow-up query plans f_{op}) is not considered. Second, a differentiation between different variants of the same type are not reflected, e.g., the difference between parallel and sequential q-gram operators. In the following, we analyze operator costs in more detail. We extend the set of considered cost factors and present how to estimate costs of complex query plans. This also helps to explain the asymptotic costs shown in Table 6.2.

operator op	m_{op}	h_{op}	f_{op}
op^{LOC}	0	0	$\lceil\lceil f_{in}\rceil\rceil$
op^{SYNC}	f_{in}	1	$\max(1, \lceil rev_{op} \cdot f_{in}\rceil)$
$\xi^{SEQ}, \omega^{SEQ}, \sigma^{SEQ}$	$f_{in} \cdot (m_l + \lvert p_r\rvert - 1)$	$h_l + \lvert p_r\rvert - 1$	$\lceil\lceil f_{in}\rceil\rceil$
$\xi^{RQ}, \omega^{RQ}, \sigma^{RQ}$	$f_{in} \cdot \lceil\lceil \lvert p_r\rvert \cdot m_l\rceil\rceil$	$\lfloor\lfloor h_l\rfloor\rfloor$	$\lceil\lceil f_{in} \cdot \lvert p_r\rvert\rceil\rceil$
$\omega^{HASH,SEQ}$	$\lvert p^{oid}_{T_{in}}\rvert \cdot m_l$	$\lvert p^{oid}_{T_{in}}\rvert \cdot h_l$	$\lceil\lceil f_{in}\rceil\rceil$
$\bowtie^{HASH,SEQ}$	$\lvert p^{a}_{T_{in}}\rvert \cdot m_l$	$\lvert p^{a}_{T_{in}}\rvert \cdot h_l$	$\lceil\lceil f_{in}\rceil\rceil$
$\omega^{HASH,PAR}, \bowtie^{HASH,PAR}$	$\lvert T_{in}\rvert \cdot m_l$	h_l	$\lceil\lceil\lvert T_{in}\rvert\rceil\rceil$
$\sigma^{QG,SEQ}$	$f_{in} \cdot \lceil\lceil (d+1) \cdot m_l\rceil\rceil$	$\lceil\lceil (d+1) \cdot h_l\rceil\rceil$	$\lceil\lceil f_{in}\rceil\rceil$
$\sigma^{QG,PAR}$	$f_{in} \cdot (d+1) \cdot m_l$	h_l	$\lceil\lceil f_{in} \cdot (d+1)\rceil\rceil$
$\bowtie^{HASH,SEQ}$ $(\xi^{QG,SEQ})$	$\lvert p^{a}_{T_{in}}\rvert \cdot \lceil\lceil (d+1) \cdot m_l\rceil\rceil$	$\lvert p^{a}_{T_{in}}\rvert \cdot \lceil\lceil (d+1) \cdot h_l\rceil\rceil$	$\lceil\lceil f_{in}\rceil\rceil$
$\bowtie^{HASH,PAR}$ $(\xi^{QG,SEQ})$	$\lvert T_{in}\rvert \cdot \lceil\lceil (d+1) \cdot m_l\rceil\rceil$	$\lceil\lceil (d+1) \cdot h_l\rceil\rceil$	$\lceil\lceil\lvert T_{in}\rvert\rceil\rceil$
$\bowtie^{HASH,SEQ}$ $(\xi^{QG,PAR})$	$\lvert p^{a}_{T_{in}}\rvert \cdot \lceil\lceil (d+1) \cdot m_l\rceil\rceil$	$\lceil\lceil\lvert p^{a}_{T_{in}}\rvert \cdot h_l\rceil\rceil$	$\lceil\lceil\lvert T_{in}\rvert \cdot d + 1\rceil\rceil$
$\bowtie^{HASH,PAR}$ $(\xi^{QG,PAR})$	$\lvert T_{in}\rvert \cdot (d+1) \cdot m_l$	h_l	$\lceil\lceil\lvert T_{in}\rvert \cdot (d+1)\rceil\rceil$

Table 6.3: Costs of physical operators

We expect all needed values to be available at each peer. In a real-world environment some of these values will have to be approximated due to limited local knowledge. As in any other query optimizer, the point is not to accurately determine costs, but to determine accurate relations between different alternatives. Table 6.3 summarizes the formulas used for estimating the costs of the main physical operators.

Intuitively, we can focus on routing operators, because local operators do not generate any messages nor bandwidth consumption. But, as described in Section 6.4, empty query plans should not be forwarded. Thus, local operators may influence the number of follow-up query plans, e.g., if no matching partners are found when processing a join operator. Without statistics about data distribution, we cannot predict the number of "dropped" query plans. We have no other choice than using the worst-case assumption that each input query plan is also forwarded. This applies to almost all physical operators. Nevertheless, we indicate if f_{in} is an upper limit for f_{op} by using the special notation $\lceil\lceil f_{in}\rceil\rceil$. f_{in} is set to 1 for the first operator that is processed in a query plan.

Synchronizing Operators

For synchronizing operators each input query plan has to be routed to the synchronization peer, which is identified by a unique ID. Thus, query plans can be sent directly without costs for lookups. Since we do not respect empty plans in f_{in}, this is an accurate limit for $m_{op^{SYNC}}$. We result in only h_l hops, because independent M^2QP copies of an original query plan are processed in parallel. $f_{op^{SYNC}}$ depends on the way the synchronizing operator handles input query plans. If, for instance, a new revision is generated after every second input plan, the operator will produce $0.5 \cdot f_{in}$ follow-up query plans. If it is totally blocking, i.e., only one final revision is generated, rev_{op} will be 0. Note that blocking increases the query answer times, which is not reflected by the number of hops here. If required, rev_{op} could be used to integrate this into the cost-based optimization.

Inter-operator parallel processing of binary operators like $\bowtie$ and $\cup$ basically results in no routing costs, because these operators are executed like local operators. The difference lies in the number of needed synchronization messages. To estimate this number, we follow the same approach as for other synchronizing operators like φ^{SYNC}. This means, we use the values f of all child operators. f_{in} is set to 1 for the first operator of each branch.

Basic Routing Operators

For range-based routing operators, we have to effort m_l messages to reach the first peer of a range r. Using a sequential operator, we need $|p_r| - 1$ messages to query the remaining peers of the range, assuming that messages between neighbors can be sent directly. This is a realistic assumption if index items of the range are clustered accordingly, as it is provided by P-Grid.

The feasibility of sequential operators strongly depends on the current network state. The higher answer times and increased vulnerability to failing peers compared to parallel implementations can be deduced from the higher number of hops. The impact of parallelism is expressed in the trade-off between h and f. One advantage of sequential operators is that we can prevent contacting the same peer multiple times, e.g., by processing all triples and q-grams that are available at one peer. This saves messages and hops, because it can result in shorter sequences. As described before, this cannot be applied in the parallel variants. Thus, parallelism comes at the expense of more messages. Parallelism influences also subsequent operators. For instance, equal OIDs may be looked up for materialization by different peers. But, this cannot be included accurately in the cost estimations, because the needed knowledge is not available a priori. An option is to include according probabilities. We indicate this fact by using the $\lceil\lceil \ldots \rceil\rceil$ notation. For instance, the messages needed for processing $\omega^{HASH,SEQ}$ are at most $|T_{in}| \cdot m_l$, but can be expected to be below. Similar, the hops needed to process $\omega^{HASH,SEQ}$ will be equal to $|T_{in}| \cdot h_l$ only when all $|T_{in}|$ are contained in one single input query plan. Thus, this is an upper limit, because usually f_{in} can be expected to be greater than 1. Similar applies for range queries. Querying a range r will result in at most $|p_r| \cdot m_l$ messages. If the underlying DHT provides sophisticated implementations, the number of messages will be below. In contrast, the number of hops will be at least h_l, denoted by $\lfloor\lfloor h_l \rfloor\rfloor$. For accurate cost predictions we refer to the works proposing sophisticated range query mechanisms, such as [60]. P-Grid's shower algorithm is analyzed in detail by [188]. Note that we list a logarithmical asymptotic limit in Table 6.2.

Advanced Similarity Operators

Instead querying for the complete set of triples belonging to an attribute, q-gram-based similarity selections query for $d + 1$ q-grams in sequence or in parallel. The main impact on

performance differences between range-based operators and q-gram-based operators lies in $|p_r|$. Querying ranges gets the more expensive the more peers are responsible for it. This is, for instance, an issue for attributes with a high amount of values. In contrast, the q-gram-based operators reveal the same costs all the same how many values for an attribute exist. The hash join promises to be efficient if the used input T_{in} is rather small. The formulas for $\bowtie^{HASH}$ in combination with ξ and ω are identical. Note that Table 6.3 shows the costs for the usual case of hash joins on instance level. In other cases, $|p^a_{T_{in}}|$ has to be replaced by the according expression.

Statistical information about data distribution, distribution of index items over peers, and selectivities can help to improve the worst-case considerations. There are several conceivable approaches for collecting and managing such information. For instance, selectivity information for q-grams can be maintained in trie-based local indexes as proposed in [187]. Maintaining such information always implies additional efforts. We suggest a cost-based query optimization abstaining from detailed statistical information on data level. Nevertheless, we need basic statistical assumptions on peer level, e.g., the number of peers responsible for a given range of hash keys. As we will show in Chapter 7, maintaining such information is much cheaper than doing the same on data level. But, a few assumptions on data level are also needed, such as the number of unique values of an attribute, lower bounds, and upper bounds. With these values we can make some basic approximations. Selectivities can be approximated on the assumption of a uniform distribution as known from traditional DBMS. They are needed for all operators depending on $|T_{in}|$, such as hashing-based operators. Distribution of triples is, for instance, needed to approximate $|p^{oid}_{T_{in}}|$ in sequential hash-based materializations. While $\omega^{HASH,PAR}$ uses a separate lookup for each input triple set, the number of lookups for $\omega^{HASH,SEQ}$ can be limited by the total number of key partitions P in the system (i.e., leafs in the P-Grid tree). Thus, $\min(|T_{in}|, P)$ can be used as a very rough limit for $|p^{oid}_{T_{in}}|$ if no further information is available. Of course, P usually has to be approximated as well. For (similarity) joins on values of an attribute `A`, $|p^a_{T_{in}}|$ can be limited by $|p_{r_{\mathtt{A}}}|$. Similar can be applied for materializations if all OIDs are indexed on only a subset of all DHT nodes. Note that this is not provided if the OID index is built as introduced in Section 4.2. Since we use $|p^a_{T_{in}}|$ and similar, the integration of extended selectivity information is possible without discarding the presented cost model.

Ranking Operators

The costs of other advanced operators can be deduced from the costs listed in Table 6.3. The top-N operators φ^{SEQ} and φ^{PAR} apply range query processing as, for instance, ξ^{SEQ} does. Thus, they result in the same values for m, h, and f as the corresponding extraction operators. The difference is that possibly several rounds have to be initiated. It is difficult for an optimizer to predict the resulting number of rounds c_{rq}. The algorithms are designed to need only one range query phase in most cases. Thus, the optimizer should expect $c_{rq} = 1$. But, this is not guaranteed. To relax this assumption, a probability of re-initiating range queries can be included in the cost formulas. After each range query phase, intermediate results have to be sent to the synchronization peer. Following, the general costs C for φ^{SEQ} on a numerical nearest neighbor query are:

$$C_{\varphi^{SEQ}} = \underbrace{m_l + C_{\xi_r^{SEQ}}}_{(I)} + \underbrace{c_{rq} \cdot 2 \cdot m_l}_{(II)} + \underbrace{(c_{rq} - 1) \cdot 2 \cdot m_l}_{(III)}$$

Part (I) refers to the costs for contacting the peer responsible for the top-1 value and querying the range r that contains the top-N values. These costs are independent from c_{rq}. But, the

higher c_{rq}, the more synchronization messages are needed. This is reflected in part (II). In each round, two subranges are queried separately. The last peer from each range sends one plan copy to the synchronization peer. Further, with each re-initiation according messages have to be sent to the first peer of each queried subrange. For the first range query this is not needed, because it is initiated by the peer responsible for the top-1 value. The peers responsible for the subranges are guaranteed to be neighbors of that peer. This is reflected in part (III) of the formula. The costs have to be modified accordingly for `MIN` and `MAX` (only one queried subrange) and other specific situations, such as finding all top-N values at the peer responsible for the top-1 value. Similar modifications are needed for φ^{PAR}, which needs more synchronization messages. Further, $C_{\xi_r^{SEQ}}$ is replaced by $C_{\xi_r^{PAR}}$ and part (III) does not apply. Costs for top-N queries on strings can be deduced in analogy.

Costs for the skyline operators can be determined following the same principle. In the different phases, existing physical operators are used. The complete costs result from the single costs of these operators plus the needed synchronization messages. The number of synchronization messages can be deduced from the follow-up plans f generated by each used operator. An important observation from [104] is that the number of messages in the parallel and hybrid variants depends on the number of peers processing local skyline queries, whereby the sequential variant depends on only the number of skyline attributes. This symbolizes the message overhead caused by the applied parallelism. [142] provides a much more detailed analysis of skyline query costs, including best-case, worst-case, and average-case approximations.

Costs of Complex Query Plans

The actual costs of each operator depend on f of the preceding operator(s). This requires to respect the order of operator execution. The costs of complex query plans Q are determined using the costs of single operators:

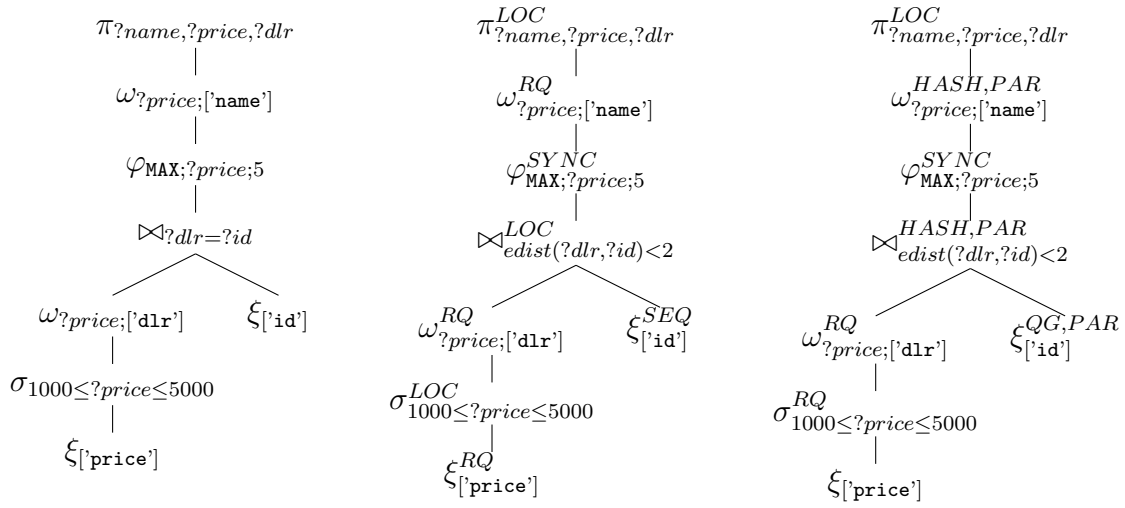
- $m_Q = \sum_{op \in Q} m_{op} + f_{root}$
- $h_Q = \sum_{op \in Q} h_{op} + 1$

This assumes that at least one plan copy is completely processed, i.e., not all copies are dropped during processing. The number of follow-up query plans of the root operator f_{root} corresponds to the estimated number of leafs in the routing graph for Q. Thus, it reflects the number of estimated replies. Each reply is sent directly to the query initiator. The final decision which plan to choose has to be based on the two factors m_Q and h_Q. It further depends on the characteristics of the network environment, user preferences, and general requirements concerning the trade-off between query answer times, robustness, and load.

variable	value
$\lvert p_{\texttt{prize}} \rvert$	10
unique values of `prize`	200
minimal value of `prize`	500
maximal value of `prize`	20500
$\lvert p_{\texttt{dlr}} \rvert$	7
$\lvert p_{\texttt{id}} \rvert$	5
$\lvert p_{\texttt{name}} \rvert$	7

Table 6.4: Example statistics needed for cost estimation

Finally, we illustrate how to determine the total costs for processing a complex query plan. We use the example from Figure 6.16. For convenience and practicability, we ignore any $\lceil\lceil \ldots \rceil\rceil$



(a) Logical query plan Q (b) First physical plan Q_1 (c) Second physical plan Q_2

Figure 6.16: Example plans for cost estimation

relations and base the cost estimations on worst-case assumptions. The logical query plan is shown in Figure 6.16(a). Figure 6.16(b) shows one possible physical query plan. Assuming statistical values as shown in Table 6.4, the costs of this plan are estimated as shown in Table 6.5. This results in total costs:

$$\begin{aligned} m_{Q_1} &= 10 \cdot m_l + 70 \cdot m_l + 70 \cdot m_l + 280 + 70 + 49 \cdot m_l + 49 = 199 \cdot m_l + 399 \\ h_{Q_1} &= h_l + h_l + h_l + 4 + h_l + h_l + 1 = 5 \cdot h_l + 5 \end{aligned}$$

operator op	m_{op}	h_{op}	f_{op}
$\xi^{RQ}_{['price']}$	$10 \cdot m_l$	h_l	10
$\sigma^{LOC}_{1000\leq?price\leq5000}$	0	0	10
$\omega^{RQ}_{?price;['dlr']}$	$10 \cdot 7 \cdot m_l = 70 \cdot m_l$	h_l	$10 \cdot 7 = 70$
$\xi^{SEQ}_{['id']}$	$70 \cdot (m_l + 5 - 1) = 70 \cdot m_l + 280$	$h_l + 5 - 1 = h_l + 4$	70
$\bowtie^{LOC}_{edist(?dlr,?id)<2}$	0	0	70
$\varphi^{SYNC}_{MAX;?price;5}$	70	h_l	$\lceil 0.1 \cdot 70 \rceil = 7$
$\omega^{RQ}_{?price;['name']}$	$7 \cdot 7 \cdot m_l = 49 \cdot m_l$	h_l	$7 \cdot 7 = 49$
$\pi^{LOC}_{?name,?price,?dlr}$	0	0	49

Table 6.5: Cost estimation for Q_1 ($rev_\varphi = 0.1$)

If we would choose to use inter-operator parallelism for the join operator, $\bowtie^{LOC}_{edist(?dlr,?id)<2}$ would be replaced by $\bowtie^{SYNC}_{edist(?dlr,?id)<2}$. This would result in modified costs as shown in Table 6.6 and in total costs:

$$\begin{aligned} m_{Q_2} &= 10 \cdot m_l + 70 \cdot m_l + m_l + 4 + 71 + 1 + 7 \cdot m_l + 7 = 88 \cdot m_l + 83 \\ h_{Q_2} &= h_l + h_l + h_l + 4 + 1 + h_l + h_l + 1 = 5 \cdot h_l + 6 \end{aligned}$$

operator op	m_{op}	h_{op}	f_{op}
$\xi^{RQ}_{['\texttt{price}']}$	$10 \cdot m_l$	h_l	10
$\sigma^{LOC}_{1000 \leq ?price \leq 5000}$	0	0	10
$\omega^{RQ}_{?price;['\texttt{dlr}']}$	$10 \cdot 7 \cdot m_l = 70 \cdot m_l$	h_l	$10 \cdot 7 = 70$
$\xi^{SEQ}_{['\texttt{id}']}$	$m_l + 5 - 1 = m_l + 4$	$h_l + 5 - 1 = h_l + 4$	1
$\bowtie^{SYNC}_{edist(?dlr,?id)<2}$	$70 + 1 = 71$	1	1
$\varphi^{SYNC}_{\texttt{MAX};?price;5}$	1	h_l	$\lceil 0.1 \cdot 1 \rceil = 1$
$\omega^{RQ}_{?price;['\texttt{name}']}$	$1 \cdot 7 \cdot m_l = 7 \cdot m_l$	h_l	$1 \cdot 7 = 7$
$\pi^{LOC}_{?name,?price,?dlr}$	0	0	7

Table 6.6: Cost estimation for Q_1 with inter-operator parallel blocking join ($rev_{\bowtie} = 0, rev_{\varphi} = 0.1$)

Cost estimations for Q_2 from Figure 6.16(c) are shown in Table 6.7. Using this variant, the total costs are:

$$\begin{aligned} m_{Q_3} &= 2 \cdot m_l + 14 \cdot m_l + 80 \cdot m_l + 80 + 5 \cdot m_l + 5 = 101 \cdot m_l + 85 \\ h_{Q_3} &= h_l + h_l + h_l + h_l + h_l + 1 = 5 \cdot h_l + 1 \end{aligned}$$

operator op	m_{op}	h_{op}	f_{op}
$\sigma^{RQ}_{1000 \leq ?price \leq 5000}(\xi_{['\texttt{price}']})$	$\lceil \frac{40}{200} \cdot 10 \rceil \cdot m_l = 2 \cdot m_l$	h_l	$\lceil \frac{40}{200} \cdot 10 \rceil = 2$
$\omega^{RQ}_{?price;['\texttt{dlr}']}$	$2 \cdot 7 \cdot m_l = 14 \cdot m_l$	h_l	$2 \cdot 7 = 14$
$\bowtie^{HASH,PAR}_{edist(?dlr,?id)<2}(\xi^{QG,PAR}_{['\texttt{id}']})$	$40 \cdot 2 \cdot m_l = 80 \cdot m_l$	h_l	$40 \cdot 2 = 80$
$\varphi^{SYNC}_{\texttt{MAX};?price;5}$	80	h_l	1
$\omega^{HASH,PAR}_{?price;['\texttt{name}']}$	$5 \cdot m_l$	h_l	5
$\pi^{LOC}_{?name,?price,?dlr}$	0	0	5

Table 6.7: Cost estimation for Q_2 ($rev_{\varphi} = 0$)

Since Q_2 contains several hashing operators, we have to estimate selectivities. This is based on the assumption of a uniform data distribution. It is not needed for the last ω operator, because the preceding φ is blocking and produces only one query plan containing the top-5 triple sets. An interesting observation is that synchronization operators are suited to reduce the number of independent plan copies. This holds also for hashing operators, because $f_{op^{HASH}}$ is independent

from f_{in}. But, this can also result in the opposite effect. Nevertheless, such operators can be used to reduce the impact of parallelism on the number of messages. This fact should be regarded in an according optimizer. We finish the discussion of physical query optimization with the example from above.

6.6 Related Work

In this section, we present the most important works related to the distributed query processing approaches that we discuss. We summarize them according to the similarities they show to our work.

General Processing Approaches

General strategies for distributed query processing, such as data shipping and query shipping, are surveyed by [120]. In the data shipping approach all data are fetched from remote sites and processed locally. This implies high communication costs and does not distribute query load. Nevertheless, it is a feasible approach for frequently queried data when some caching mechanisms are used. Query shipping fits better for distributed processing. In this approach, (sub)queries are shipped to where data reside and are processed remotely. Compared to data shipping, this usually results in lower communication costs. Further, query load is distributed due to applied concurrency and parallelism. Query shipping is used in most distributed DBMS. [120] mentions also hybrid shipping, which combines data and query shipping depending on the query and its state. This requires more complex query planners. The M^2QP approach is based on query shipping, but integrates facets of data shipping and code shipping to enable autonomous handling of query plans. Though [120] assumes a globally known data schema, it provides a well-founded basis for distributed query processing approaches.

The idea of MQPs is presented in [163,162]. Queries are processed by shipping (sub)queries that additionally contain data and data references. Data references are resolved by routing to the responsible peers, which replace the query operators by according results. This combines data and query shipping and is called combined shipping by the authors. In the original MQP approach, a single query plan is initiated. Each receiving peer mutates the plan and forwards it, which results in a sequential processing. The state of a query is completely encapsulated in the plan, which implicitly enables adaptive query processing. Sequential processing bears the characteristic that the resources of only one peer are occupied at a time. According to the authors, this gains site autonomy, flexibility, and reduced coordination efforts. But, to use the full benefits of query shipping, the resources of multiple peers should be occupied at a time. A higher degree of parallelism and concurrency distributes load and is better suited for fragile and unreliable environments. For future work, the authors suggest so-called *mutant strains*, which conceptually are parallel MQPs that require synchronization at a later time.

In our work, the MQP concept is extended to meet modern mechanisms and primitives for large-scale distributed query processing. An interesting work suggesting a couple of these new mechanisms is [183]. The stateless M^2QP approach and the physical operators in general utilize chaining and referral, extended by delegation and forwarding. Parallel approaches additionally utilize parallelism and concurrency in processing. A general overview of related techniques from P2P systems is provided by according surveys, such as [175, 129]. General principles of distributed query processing at large scale are also matter of [36]. Principally, ObjectGlobe applies a query shipping approach. Participating nodes are classified into data providers, function providers, and cycle providers. Data are obtained from data providers, operators are executed at cycle providers, and external query operators are received from function

providers. The authors clarify that an according query optimization specifically designed for distributed systems is mandatory. But, the ObjectGlobe system depends on a central directory and a central optimization process. A strength of this work is the integration of security issues into the complete process of querying. [112] proposes an implementation of query processing for Deep Web services. A distributed processing approach on the basis of query shipping is applied. In the proposed dynamic market places, distributed queries sit behind hyperlinks and are processed using the QueryFlow system. Supporting distributed queries without the need of explicit user knowledge presents a first step towards enabling the mechanisms that are mandatory for Public Data Management.

P2P Database Approaches

A similar extension to the MQP approach as we propose is used in PeerDB [156, 153], which is based on BestPeer [152]. PeerDB also proposes the adoption of database functionality on top of DHT overlay systems. The utilized query agents are comparable to M^2QPs. Similar, executable code is shipped to the peers where relevant data reside. The agents are able to manipulate the shipped query plans. They also collect statistics about the network, which are used to reorganize it in order to increase query performance. PeerDB supports only full-fledged databases as endpoints. It also supports querying with limited knowledge about schemata. In order to achieve this, managed relations are automatically annotated by semantic metadata motivated by IR approaches. These data are handled in the same way as raw data. But, integration into query processing follows a two-phase approach and is not as stateless and tightly integrated as proposed by us. In the first phase, candidates are determined, which are contacted in the second phase. This can result in long query times and is problematic in the presence of unavailable resources. In analogy to Napster [144], PeerDB uses centralized directory servers. This is a significant problem for achieving high scalability and should be replaced by totally decentralized lookup mechanisms. Further, query processing is based on a flooding approach similar to the Gnutella [74] way of processing queries.

Another work that aims for supporting complex database-like queries in DHT-based distributed storage systems is PIER [100, 99]. It has been implemented on top of CAN [169], CHORD [196], and recently Bamboo [173]. PIER supports query processing only on homogeneous data. But, the underlying concepts of query processing are very similar to those of our work. The authors analyze requirements and solutions strictly from the perspective of distributed relational DBMS. Range queries are supported on the basis of prefix hash trees [170], which follow the same principles as P-Grid, and a broadcasting mechanism. Different routing strategies are not discussed as detailed as by the work in hand. In contrast, specific operators like joins are handled in more detail. Beside joins, PIER supports selections, projections, group-by, tee, union, and duplicate elimination. All queries can be run as snapshot queries or as continuous queries. The supported operators make excessive use of rehashing (parts) of relations. For instance, equi-hash-joins are processed by hashing both input relations using the join attribute (if the index does not already exist). We consider this as too expensive and infeasible for Public Data Management scenarios. Further, the system is (up to this point) not able to provide any guarantees on running queries, such as estimates for query completeness. The suitability of PIER for file-sharing applications, in which no schema heterogeneities exist, is shown in [131]. In that work, PIER is used as an index for rare files in the Gnutella network. This reduced the rate of empty results by about 18%.

Similar frameworks that provide database-like features on top of DHT systems are proposed by [206, 24]. [206] hashes tuples in a similar way as PIER does. But, they also apply order-preserving hash functions to support efficient range queries. To leverage performance of range

queries, so-called range guards are introduced. Conceptually, these range guards apply caching of frequently queried attributes through rehashing. This limits the asymptotic runtime to the number of range guards rather than to the number of participating peers. Complex queries are based on rudimentary queries, which include equality selections, range queries, joins, and grouping. The system supports only numerical data, which is a significant disadvantage for most Web-based applications. Since indexing each attribute results in considerable overhead, the authors suggest to store only references. P-Grid, the DHT used in this work, is also based on order-preserving hashing. In contrast to [206], it provides load-balancing even for skewed data distributions [7]. Further, [206] supports no queries on schema level, no similarity queries, and lacks an evaluation. [24] proposes methods for range queries and advanced query types on DHT overlays. The authors also suggest to choose between multiple indexes for query processing. A kind of database catalog is built using the functionality of the underlying DHT. Queries are processed like sequential MQPs. The system supports only simple subqueries that can be combined by conjunction and disjunction. Further, processing is done in phases: the first phase for collecting catalog information, the second for building subqueries, and the third for finally collecting resources. Range queries are implemented using a range search tree, which is comparable to aggregation trees and suffers from bottlenecks and dependencies between peers. As the most approaches discussed here, the work lacks discussion about guarantees and predictability, such as estimating query completeness.

A work that follows the same approach as PIER by horizontally distributing fragments of relations is [185]. That work uses CAN as the underlying DHT. The authors propose a physical query algebra very similar to the one presented in this work. The algebra includes projections, selections, range queries on numerical data, string similarity queries, and different join and grouping variants. Similar to PIER, the authors make use of rehashing to process certain operators. In one distribution strategy, a locality-preserving hashing along a space-filling Z curve is applied, which enables efficient range queries. Similar to the work in hand, a query shipping approach based on MQPs is used. The proposed physical query algebra is a first step towards Public Data Management scenarios, but still all participants have to agree on a common schema. The work lacks a real-world evaluation and the support of schema queries, top-N queries, and similarity joins.

The idea of choosing between different distributed indexes during query processing is also used in [214]. In contrast to our work, the authors propose to generally chooses only a subset of data to be indexed. The indexes are used to speed up query processing, but are not mandatory for locating data. CHORD-based DHTs as index structures are also exploited in [31]. But, the introduced system AmbientDB is not a pure DHT. DHTs are used only for a subset of rather simple queries, such as equi-selections on the index key. A similar approach based on CHORD is used for indexing XML path expressions in [71]. Consequently, only path queries are supported. Further, evaluation is based on only a simulation.

Query Processing on Triple Data

RDFPeers [41] provides a storage layer very similar to the one proposed here. Queries from formal query languages, such as RDQL [140], can be mapped to the supported native queries. RDFPeers supports only exact-match queries, disjunctive queries for sets of values, range queries on numerical data, and conjunctive queries for a common triple subject. The authors focus on typical RDF queries and the implicit graph-based query evaluation. Sophisticated database-like query processing capabilities and heterogeneities are not considered. In contrast to our approach, query resolution is done locally and iteratively. Most RDF query engines follow such a centralized approach based on data shipping. In contrast, DARQ [166] is a federated

query engine for SPARQL queries. Queries are decomposed into subqueries and shipped to the corresponding RDF repositories. DARQ uses query rewriting and cost-based optimizations. It strongly relies on standards and Web service descriptions and can use any endpoint that conforms to the SPARQL protocol. Further, it does not require an integrated schema. DARQ supports standard query operators and two join variants on bound predicates. All query processing is designed to explicitly match the graph-based RDF queries. Further, query planning is static and centralized, utilizing the service descriptions that replace any indexes. Subqueries are distributed using a kind of broadcasting mechanism. This makes DARQ infeasible for the scenarios that we propose. In [88,87] the whole RDF model graph is mapped to nodes of a DHT. DHT lookups are used to locate triples. This implements a rather simplistic query processing based on query graphs that can be mapped to the model graph. Query processing basically conforms to matching query graph and model graph. RDF-Schema data are indexed in addition and used by applying RDF-Schema entailment rules. Query processing is two-phase. First, candidates are determined by querying schema knowledge and knowledge about resources. After combining these distributed RDF graphs, dependencies between variables and triples are resolved and finally result data are gathered. Sophisticated query constructs that leverage the expressiveness of queries are not considered.

Similarity Queries

Similarity operations are an extremely important and essential part of a universal storage system. They have been considered in, for instance, [25] and [204] before. Both works lack details and an extended experimental evaluation. Several other approaches exist which consider range queries as similarity queries. However, we consider here only systems that support a more complex notion of similarity, such as top-N queries and similarity on string data. LSH forest [25] uses a locality-sensitive hashing (LSH) function to index high-dimensional data for answering (approximate) similarity queries. The queries return the m points in the data set closest to the query according to a distance function. The system is based on P-Grid and stores documents in the overlay network using the LSH function. Therefore, similarity queries can be performed by first routing to the peer closest to the initial query and then returning similar documents by using existing neighbor links. The paper does not provide an evaluation of required messages or bandwidth consumption. EZSearch [204] is based on the Zigzag hierarchy which clusters semantically close nodes in a multi-layer hierarchy. It supports range queries and top-N queries. The evaluation of the system by a simulation shows that the system works well for both query types even for Zipf-like query distributions. But, it remains unclear how the system deals with skewed data distributions, which require sophisticated load balancing mechanisms. Additionally, no experimental evaluation exists. The mentioned works utilize the idea of building semantically clustered P2P systems, which cluster related data on neighboring peers for supporting similarity. This is also used in several IR proposals that apply typical IR measurements like the vector space model. An example is [201]. [22] introduces a complete family of access methods for such similarity queries, called SWAM. The authors propose to build according small-world overlays using Voronoi-based techniques. Thus, the approach cannot be directly mapped to DHT-based systems. But, the process of similarity search is closely related to the top-N algorithms proposed in this work. First, a query is routed to the center of a range. Afterwards, queries are shipped to the peers in the near range to answer queries. This strongly relies on the clustering feature of the overlays and is comparable to the way we utilize range queries. A more flexible mechanism, comparable to our suggestion for processing string similarity, is not considered in [22]. Beside range queries and top-N queries, SWAM only supports exact match queries. The authors further suggest to use a metric based on query times

for optimization. As this is based on average answer and message passing times, this actually corresponds to the number of hops a query needs. Compared to all these works, our approach supports a larger set of query types on structured data, i.e., top-N queries with different ranking functions, similarity joins on schema and instance level as well as other advanced similarity queries, and allows us to combine these with standard relational algebra operators. In conjunction with P-Grid, it works efficiently on a wide range of data distributions and network conditions.

String similarity is exhaustively researched for centralized database systems [148,77]. There exist several works considering string similarity in widely distributed systems. Some also propose solutions based on the idea of q-grams. [187] discusses selectivity estimation for q-samples in the context of Web-based information integration. That work motivated our approach for q-gram-based similarity evaluation. Q-grams are also considered in [218], but the authors argue that they are not suited for DHTs. But, they also agree on the importance of handling spelling errors, which they motivate by increased precision and recall in file-sharing applications. They propose to base string similarity on soundex indexes, another form of expressing string similarity. We showed that q-grams are suited for efficient processing of string similarity in DHTs as well.

Ranking Queries

Mapping similarity to range queries is very similar to a lot of top-N query proposals. There exists a wealth of proposals dealing with such queries in centralized databases. Popular approaches are based on mapping top-N queries to selection queries using existing indexes [40] and to use materialized views [96]. [43] presents the three phase uniform threshold approach designed for distributed systems, an enhancement of the popular TA approach [67]. A central instance has direct access to sorted lists of attributes, which are distributed over the nodes of the system. In multiple rounds the central instance computes partial aggregates that serve as lower bound estimates. These bounds are used to reduce the number of candidates that are queried from the sources. Beside the sorted access, random access is needed as well in order to completely materialize final objects. A major drawback is the need for sorted access. There are suggestions how to handle sources that do not provide such functionality [136]. Nevertheless, the round-based character of the approach is mandatory. For answering a single query the network has to be flooded three times. Our approach tries to achieve the same by querying only peers in a guessed range, which is hopefully finished after the first round. [21] presents techniques to improve performance for repeated top-N queries. Statistics are collected using a kind of feedback approach and used for optimizing the processing of identical queries later on. For the first processing, a multi-round algorithm is applied, too. Further, the idea suffers from consistency issues typical for caching approaches, especially in very dynamic and fragile networks.

Skyline queries are intensively researched in the database community [33, 122, 161, 20, 213]. Several approaches were designed to decrease the overhead of computing skylines in centralized databases. But, approaches like Block-Nested-Loops (BNL) [33], Divide-and-Conquer [33], Nearest-Neighbor [122] or Branch and Bound Skyline (BBS) [161] cannot be adopted directly to widely distributed systems. Processing skylines in distributed systems gains more and more attention. [97] introduces skylines for mobile ad-hoc networks and [128] for cluster-based P2P systems. Despite the fact that both works aim for minimizing communication costs, they are based on different assumptions and fundamentals compared to our work. [20] proposes an algorithm that exploits the TA principle [67] of sorted lists for processing skylines in Web information systems. There are several major drawbacks. First, the approach requires a specialized

network structure in which the processing node has direct access to all other nodes. Second, sorted lists of all objects are required. And third, it would neither scale with the number of nodes in the network nor with the number of skyline dimensions. Further, all computations are done locally, which neglects the possibilities of distributing load. Efficient skyline processing for unstructured P2P systems, enriched by the option to determine *relaxed* skylines, is introduced in [94]. This approach utilizes specialized spatial index structures and was designed for PDMS. Further, the authors extend the dominance relation to skyline regions, which are used to narrow the search space of skyline candidates. Relaxed skylines are suited to lower communication costs even more. In this context, data points can be represented by other points in a certain distance. This decreases the number of peers that have to be contacted. The proposed techniques cannot be adopted straightforward, because PDMS rely on unstructured P2P systems, which apply a different storage model than DHTs do. Another work that considers skyline processing in distributed environments is DSL [213]. It progressively computes constrained skylines using a CAN built on top of the dimensions of later skyline queries. The general idea is to query the CAN regions in an order that potentially decreases the number of involved peers. This idea is similar to the dominance relation of regions presented in [94]. This helps to reduce communication costs, but on the other hand decreases the degree of parallelism. Another great disadvantage is that the approach is not general enough, because the CAN dimensions have to correspond to the skyline dimensions. [208] presents the SSP approach for skyline processing, which is closely related to the one proposed here. The SSP approach also tries to minimize the number of candidates that have to be queried. Instead using the frame defined by the extrema of all dimensions, they look up a point near the coordination origin (if only `MIN` ranking functions are used). This is because such points promise to dominate specifically large regions. For looking up the remaining candidates, an algorithm similar to P-Grid's shower algorithm for range queries is applied. SSP is implemented on top of the Baton DHT [102]. It also supports a simple method for balancing data and load. Recently [209], the authors extended their approach to achieve a higher degree of parallelism and to also work on the CAN DHT. Both approaches, SSP and frame skyline, are suited for different situations. They may also be extended by other methods for pruning the search space. In future works, several approaches should be compared in order to identify which solution is suited for which situations. Ideally, data distribution and DHT characteristics should be taken into consideration for this choice. A very comprehensive overview of central and distributed skyline query approaches can be found in [91].

Dynamic Plan Processing

A survey on adaptive query processing techniques can be found in [76]. That work summarizes the main concepts of adaptivity, which are also enabled in our framework. Eddies [18] are the incarnation of adaptive query processing. An Eddy is a kind of pipeline for tuples and is used for routing these tuples to operators based on several statistics. All operators in which a tuple can already be processed are available for this choice. The Eddy technique allows for high flexibility in selecting executing peers and distributing operators. Restructuring and re-optimizing query plans is implicitly supported. Routing strategies for distributed Eddies are discussed in [203]. In distributed Eddies, the routing decision is not made at a central instance. Rather, it is made at each operator. The operators themselves are distributed among the peers. Peers maintain statistics for operators, which can be exchanged regularly or shipped with the query plans. Eddies have been successfully implemented in, for instance, [177] and the PIER system. Our framework provides all necessary prerequisites for this as well. The Eddy principle can be

implemented by restructuring query plans accordingly at each involved peer. This is implicitly supported by the independence of M^2QPs and their inherently autonomous handling.

One of the first works proposing online query processing is [90]. The work criticizes the (up to the point of publication) traditional batch processing of aggregation queries and suggests several ways to control and observe the query execution on the fly. The work acts as an archetype for a lot of later proposals, reaching from centralized databases over distributed systems to stream processors. The concepts of peer pipelining, online, and evolving algorithms are based on this early work as well.

Integrating Semantic Correspondences

The integration and resolution of semantic correspondences is thoroughly discussed in works like [58,5]. GridVine [6,58] is a peer data management infrastructure addressing both scalability and semantic heterogeneity. Scalability is addressed by peers organized in a structured overlay network forming the physical layer, in which data, schemata, and schema mappings are stored. Semantic interoperability is achieved through a purely decentralized and self-organizing process of pair-wise schema mappings and query reformulation. This forms a semantic mediation layer on top and independent of the physical layer. The semantic layer enables peers to share data in the overlay network according to local schemata that are interlinked by user-defined schema mappings. These schema mappings are then used for automatic query reformulation, which allows queries to traverse a sequence of schemata at the mediation layer and retrieve all relevant results, irrespective of their actual schemata. This corresponds to navigating through the mapping graph. GridVine offers a recursive and an iterative gossiping approach, i.e., query reformulation approach. In iterative gossiping, the peer issuing the original query is responsible for retrieving all mappings and reformulating all queries iteratively. In recursive gossiping, the reformulation process is delegated to those peers receiving reformulated queries. Recursive gossiping performs systematically better, because it distributes the reformulation load among peers. The semantic gossiping approach was first formalized in the context of the "chatty" Web in [5]. GridVine shares several aims and features with our approach, e.g., both use P-Grid as the underlying overlay network and are able to handle data and schema heterogeneity to a similar extent. GridVine supports triple pattern queries with conjunction and disjunction, implemented by distributed joins across the network. It does not apply the idea of cost-based database-like query processing over multiple indexes. We support similarity-enriched SPARQL-like queries with in-network query execution. On the contrary, we currently do not support something similar to GridVine's verification of schema mappings to identify incorrect schema translations [57]. GridVine also deals with transitivity and cycles in schema correspondences in more detail than we do up to this point. Piazza [84] is a PDMS dealing with scalability and data heterogeneity. It is one of the precursor systems in the context of handling schema heterogeneities on logical level. Piazza uses a mapping language for defining mappings between sets of XML source nodes with different document structures (including those with XML serializations of RDF). The system uses the transitive closure of mappings to answer queries. It is able to follow mappings in forward and reverse directions and can both, remove and reconstruct XML document structure. Unlike our approach, Piazza uses central indexing. An extension to use an underlying DHT seems to be planned for the future. The integration of semantic correspondences into query processing as proposed by [117] is similar to our approach. RVL views [134] are used to describe participating sources. These views are indexed in a DHT. Resolving correspondences is based on looking up the according index items. This is implemented in rounds. In analogy to our approach, the queries resulting from that process are started as soon as enough mapping data are available. But, as rounds are initiated by the query initiator,

the method is not as flexible as the M^2QP approach, where different synchronization peers can be used and resolution of mappings is absolutely stateless. Further, the approach assumes globally agreed schemata, which are managed in according semantic overlay networks – one for each agreed schema.

Chapter 7

Guarantees and Predictability

Predictability is a crucial challenge in widely distributed and totally decentralized systems. DHT systems usually guarantee logarithmic lookup costs asymptotically, but the exact overhead cannot be predicted. It is even more difficult to predict result-quality measures, such as correctness, actuality, consistency, and completeness. Main problems arise from the lack of global knowledge, unpredictable changes in the peer population (churn), lack of global control, strictly pair-wise interaction, etc. The advantages of overlays – no global knowledge required, no single-point of failure, superior resilience against partial failure – incur new problems when it comes to supporting such advanced techniques. Methods from classical database systems can be applied only to a limited extent.

Providing information about the completeness of a (temporary) query result is one of the most important requirements. This is on the one hand very useful and convenient for the user, especially in the context of long-running queries and online query processing. To map this to the CAP theorem discussed in Section 3.1, Web users will typically prefer availability over consistency. They will prefer fast results that are "good enough" over absolutely correct and complete results that are received after long waiting times. The user has to define *what* is "good enough" – the system has to answer the question *when* this point in time is reached. This is not possible without information about query completeness. On the other hand, we introduced a couple of physical operators that can benefit from an accurate completeness estimation internally. This holds for all synchronizing operators. A perfect example are the top-N and skyline operators from Section 6.2.2. The implicit synchronization phases are main beneficiaries, concerning query times as well as accuracy. But, also inter-operator parallel processing requires synchronization. Operators should define *what* amount of result data is sufficient based on statistical correlations. User preferences should also be considered. In all cases, query completeness should be accompanied by according guarantees. Only then it can be used on a meaningful basis.

In this section, we propose a solution for approximately estimating query completeness. The approach comes along with guarantees. Negligible small information about the routing graph are shipped together with each plan copy and evaluated at the query initiator. As soon as a specific amount of information is available, the method can guarantee a 100% correctness. Before that point, it supports determining probabilistic guarantees [123]. In Section 7.1, we briefly discuss the notion of query completeness in distributed data systems in general. The key ingredients of the proposed method are introduced in Section 7.2. Parts of the whole method have to be provided by the underlying DHT systems. We present an according technique for the P-Grid system in Section 7.2.1. The description of the approach is concluded by discussing guarantees

in Section 7.3. Finally, Section 7.4 lists major related works in the field of predictability in P2P data systems with a focus on estimating query completeness.

7.1 Query Completeness in Data Systems

There exists a lot of work on completeness in the context of data quality [141, 146]. There, it is typically understood as the ratio between received result data and all result data known. This does not fit to the idea of in-situ querying and the open world assumption. Thus, we distinguish between data availability and query completeness. Data availability refers to the amount of data that is available at query time – with respect to the real world. Availability is usually provided by P2P systems. To achieve this, according replication techniques are a good choice to deal with peer failures or joining and leaving peers (called *churn*) [28].

The classical database view is based on a closed world assumption and thus, a query will always return all matching answers. In contrast to this, P2P data management is inherently open world: While processing a query, peers can fail, leave or join the network, or simply send no or a delayed answer [78]. Though this can be mitigated by replication and delay-tolerant query techniques, there is no guarantee that all answers which potentially exist can be returned. Query processing usually follows a best-effort approach. That is, data are queried in-situ, which means the result is complete when all relevant data from all peers available at query time are received. Query Completeness is defined as the ration C between the amount of data a in answers received to a query Q and the amount of data α of answers we would get if all responsible and available peers at query time t would respond.

$$C(Q,t) := \frac{a(Q,t)}{\alpha(Q,t)}$$

C is defined as above for Q and for any subquery of Q. This allows for estimating completeness of intermediate results produced during processing Q. We are not trying to guarantee completeness, neither we want to improve the functionality of the underlying DHT. By completeness estimation, we mean to provide information to the user or the system that allows to assess the quality of the answer in terms of the ratio between the size of the returned result and the result we would get if all peers would answer. Though we do not reflect churn in general with this definition, we take churn during query processing into account. If peers fail during the processing of (parts of) a query, this cannot be handled by replication techniques. To illustrate the problem, Figure 7.1 shows average query completeness for three queries, each initiated 15 times in the Planet-Lab [50] test-bed. Despite a high replication factor was used in order to handle the dynamics of the network, average result sizes are significantly below expected result sizes. The amount of missing data increases with the complexity of the query. This is reflected by the number of peers involved, which in turn is reflected by the number of replies received.

Intuitively, query completeness is understood on the *data* level. This means the amount of data a and α is expressed in numbers of data items received. Estimating C depends on accurate data summaries. Such summaries have to be build and maintained in a decentralized way. This presents special challenges, like actuality of the summaries and distributed commutation techniques. Moreover, this results in additional overhead. This can be avoided by following an approach on *peer* level. In this approach, we do not compare the received result data with the expected data (about which we have no available knowledge). Rather, we compare the number of received replies r with the number of replies ρ if all available peers reply. The number of replies corresponds to the number of replying peers. In order to provide significant results, this must be supported by a good load-balancing scheme. Like this, data are balanced among the

peers. This is based on the assumption that a linear correlation between the number of failed peers and the resulting miss of result data exists. Such load-balancing features are supported by a set of DHTs, such as P-Grid [3]. Thus, we redefine C as follows:

$$C(Q,t) := \frac{r(Q,t)}{\rho(Q,t)}$$

We believe in the peer level approach as a satisfyingly accurate method. The resulting inaccuracy – the difference between completeness on data and on peer level – is countervailed by the overhead reduced by far. As we will show, accurate completeness estimation on peer level can be achieved with only minor additional bandwidth overhead and in parallel to query processing. The main task for implementing query completeness is estimating a good value R for the expected number of replies ρ at any time t.

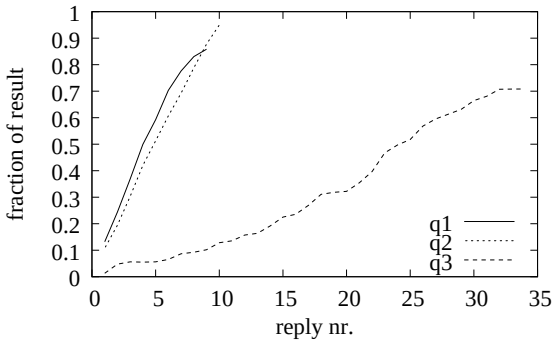


Figure 7.1: Result quality in Planet-Lab

7.2 Estimating Query Completeness on Peer Level

The processing of a single query plan Q is started at the query initiating peer. Processing this query results in parallel plan copies $Q_i, i = 1, \ldots, \rho$, where ρ is the number of all plans generated. Each single plan is processed separately and results in a reply sent to the initiating peer. To estimate the query completeness, we have to estimate a value R for ρ. For convenience and eased understanding, we assume that each plan copy results in at least one reply. This corresponds to continue processing plan copies with empty intermediate results. As such plans should be dropped, the process has to be modified slightly. Further, we assume that one logical operator is replaced by the same physical variant by each involved peer. This can be achieved by static query planning. Autonomous dynamic planning and other forms of adaptive query processing require modifications as well. We indicate the straightforward modifications at the corresponding places.

We aim at an incremental and online refinement of the estimation similar to the idea of progress indicators [49]. In contrast to this idea, our approach provides information about the completeness of the query answer, but not primary about the progress of the query execution. There are several approaches for estimating query completeness that are conceivable. One idea is to initiate a kind of probing queries. Before initiating a query or a new operator, such queries could be used to identify all peers that will be involved into processing. This means, we have to simulate the processing of the plan, e.g., by querying the range of an involved attribute. Based on this information, the number of expected replies can be determined exactly during actual

query processing, as long as no churn occurs. But, operators that depend on input data, such as hash-based operators, cannot be simulated at the query initiator. Besides this, there are several obvious drawbacks: First, this technique corresponds to a separate query phase, which results in resource consumption and waiting times. Second, missing data due to unavailable parts of the DHT topology are not reflected. Last but not least, the state of the network could change between the probing phase and the plan processing phase. Another idea is to use a kind of heartbeat mechanism. Each peer involved in processing an operator can signalize the state of the query. This can be used to monitor the execution of the query as well as the completeness of the query result. As heartbeats imply additional messages and may get lost as well, a method for completeness estimation should not rely on them. The approach presented here is also based on logging the process of query execution, but does not require additional messages. A similar mechanism based on *Milestone Messages* (MiMes) is introduced as an optional technique for increasing the estimation quality in early states of a query.

Despite the wide variety of processing strategies and physical operators, we identified only a handful of main differences among these techniques relevant for applying an accurate completeness estimation. Our approach of completeness estimation builds on the same principle as estimating the total number of plan copies in the cost model, see Section 6.5. As introduced, the number of leafs in the routing graph corresponds to the number of replies. Routing graphs have been used for illustrating physical operators and query plan processing strategies. But, using the output of the cost-based planning is not sufficient for estimating query completeness. This is mainly because costs are estimated a priori in a planning approach. Since the relative differences between operators are of primary importance, guarantees are not mandatory. This is different for estimating query completeness, which should follow a control and monitoring approach based on facts rather than (maybe wrong) statistics. Further, for enabling adaptive query processing, cost-based optimizations and modifications can be handled autonomously at each peer. The general idea of our approach is to gather information about the different states of a query in each plan copy. This information is used at the query initiator to estimate the query routing graph. Following, according information is extended with each new reply. The crucial point is to approximate the parts of a routing graph for which no exact knowledge is available. In the basic approach, for each subtree we know that exists, but from which we did not receive any reply yet, we assume at least one reply.

In the following, we first summarize and formalize the notion of routing graphs and present according classes of routing strategies. Afterwards, we present the general algorithm for estimating query completeness, discuss main ingredients, and introduce a few optimizations.

Routing Graphs

The paths that all plan copies Q_i belonging to the same query Q take through the network can be illustrated as a directed tree, with the leafs corresponding to the set of peers sending final replies. Thus, the number of leafs in that tree is ρ. This tree is the query routing graph $RG(V, E)$. A vertex in the graph, called *routing point rp*, represents a peer responsible for processing a part of any Q_i, i.e, the peer is involved in the processing of at least one operator contained in Q. Edges are called *routing connections*. For each plan copy that is routed from one routing point to another we have one such edge in RG. There exists a connection $rp_x \rightarrow rp_y$ iff a plan copy is sent from rp_x to rp_y. A *routing level* is crossed if this is due to starting the processing of a plan operator. In the case of plan forwarding in sequential operators we say we stay on the same routing level. Thus, each routing operator corresponds to one designated routing level. Between two levels, multiple local operators may be processed A peer may correspond to several vertices of a routing graph, because it may be contacted several times

for processing different parts of Q. A routing edge may span over several intermediate routing peers, which is solely due to the DHT. As a consequence, the query routing graph is a topology overlaid on the topology of the DHT overlay (ring, tree, etc.) and the graph also differs from its query plan Q.

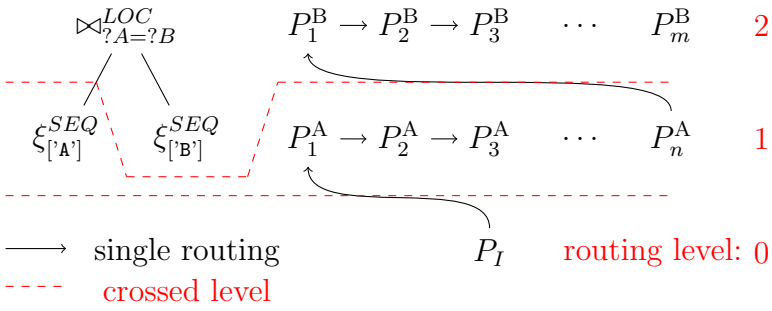


Figure 7.2: Sequential processing

Figure 7.2 shows an example routing graph for sequential processing. $\xi_{['\mathtt{A}']}$ extracts all triples corresponding to attribute `A`. Assuming post-order processing, after extracting all data for attribute `B` the final join $\bowtie_{?A=?B}$ can be processed on P^B_m. The right side of the figure shows the routing graph. The query is initiated at the query initiator P_I, which sends the plan to the first peer responsible for `A` (P^{A}_1). Thus, there is only one routing connection from routing level 0 to level 1. Each peer responsible for a part of `A` extracts local data, includes it into the query plan, and forwards the query accordingly. After all n peers in this sequence were contacted, the nth peer crosses the next routing level by routing the plan to the first peer responsible for processing $\xi_{['\mathtt{B}']}$. After the single plan copy has passed the sequence $P^{\mathrm{B}}_1 \ldots P^{\mathrm{B}}_m$, the join can be processed finally at P^{B}_m. This is the peer sending the one final reply to the initiator. Intuitively, estimating the number of replies for strict sequential processing is trivial, as

$$\rho = 1$$

always holds. Peer pipelining can be integrated similar to the parallel processing strategies discussed in the following.

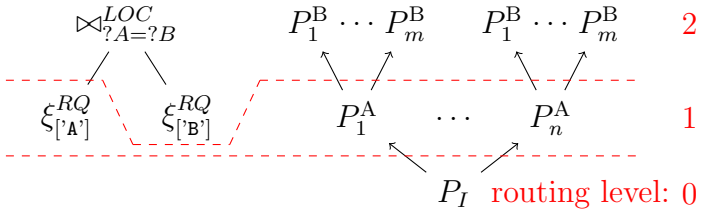


Figure 7.3: Intra-operator parallel processing

Figure 7.3 shows the routing graph for the same query plan, but with intra-operator parallelism. First, all peers responsible for `A` are contacted in parallel. Each of the peers P^{A}_i independently contacts all peers responsible for attribute `B` to finalize processing. Note that the peers corresponding to P^{B}_j connected to P^{A}_1 are the same as those contacted by P^{A}_2, P^{A}_3, and so on. But, each plan is processed independently. Intuitively, the number of final replies is

$$\rho = n \cdot m \quad .$$

In general, for strict intra-operator parallel processing based on range queries the number of replies is

$$\rho = \prod_{l=0}^{L-1} d_l \quad ,$$

where L is the maximal routing level and d_l denotes the fanout at each level l. For also concerning data-dependent intra-operator parallelism, e.g., in hash-based operators, the number of leafs can be described by a recursive formula:

$$p_0 = 1 \tag{7.1}$$

$$p_l = \sum_{k=1}^{p_{l-1}} d_{(l-1),k} \qquad l = 1, \ldots, L \tag{7.2}$$

$$\rho = p_L \tag{7.3}$$

where p_l is the number of vertices at level l and $d_{l,k}$ denotes the fanout of the kth vertex at level l.

Routing Methods

By analyzing the routing graphs for different processing strategies, we observed that the different strategies can be classified by only two relevant routing methods. The crucial difference is whether a peer starting a routing knows how many peers will be contacted (i.e., the out-degree of the corresponding routing point is known a priori) or not. Following, we distinguish:

1. Dynamic Overlay Routing (DOR): The out-degree is not known a priori. Usually, this is done by addressing a certain key space and letting the overlay decide how to forward the separate plans to all peers in this key space. In most cases, this means that the routing of plans is independent of already determined data. We apply this concept in all range-query-based operators. The number of peers in a queried range is not known, though the range itself is.

2. Fixed Number Routing (FNR): This class of routing methods covers all processing strategies in which the peer starting a routing knows the number of generated plan copies. In most cases, the routing of plans is dependent of intermediate input data. This corresponds to all hashing-based operators.

In the following, we base our explanations on these two classes of routing methods. Note that all of the introduced physical operators can be classified accordingly. For instance, a sequential operator corresponds to an FNR. In the routing graph, this is reflected by a fixed out-degree of 1 at each routing point. Special cases, such as a mixture of both approaches (e.g., issuing a fixed number of range queries depending on the input data) or simplified routing methods (e.g., if the number of peers contacted in one processing step is already known at query planning time) can be integrated easily. Throughout the remainder of this section, we describe how complex query plans that combine these routing methods can be estimated. Following, all query plans that are possible on the basis of the introduced operators are supported by the proposed approach for completeness estimation.

General Idea

Based on the notion of routing graphs, the basic idea of estimating query completeness works as shown in Algorithm 17. This procedure is called every time a new reply arrives at the

Algorithm 17: Basic algorithm run with each received reply

```
1 update-overlay();
2 update-RG();
3 r = r + 1;
4 R = 1;
5 forall routings x ∈ RG → get-levels() do
6 |   R = R + x → estimate(R);
7 end
8 return r/R;
```

initiator. For estimating DORs, we have to maintain some information about the structure of the overlay. Details on this are presented in Section 7.2.1. After updating this information and the estimated routing graph RG (one for each initiated query), the fraction of received results r and estimated final replies R is returned. Thus, a currently estimated completeness (on peer level) is determined in an online fashion. R is estimated by iterating over all levels of RG and applying Formulas 7.1 to 7.3.

Routing Trace

But why is it adequate to differ only between two routing methods, if there is such a wide variety of possible processing strategies and indexes to be used? The point is the amount of information we can include into each plan copy. This information is received at the query initiator with each single reply. We can integrate information about the way the plan copy took when traveling through the network. As a consequence, this *routing trace* allows to identify the one path from the query initiator to a leaf of the routing graph that corresponds to this reply. Intuitively, for estimating the number of replies a query results in, the quality of this estimation depends on the kind of information about each routing point we can extract. For instance, if we know the out-degree d of a routing point before starting that routing, we can include this information in each of the d resulting routing traces. Thus, this information is available at the query initiator as soon as the first of the corresponding plan copies is replied. This holds for FNRs. For DORs we always know the addressed key space, but not the exact out-degree of the initializing routing point rp, unless all replies from the subtree rooted by rp arrived. The crucial point is to estimate the parts of the routing graph that are not yet known.

This indicates an advantage of sequential operators in conjunction with peer pipelining. Results are returned early, similar to the parallel processing. But, the out-degrees of routing points in the sequence are known, even if the corresponding routing traces are not yet received. This is different for parallel operators. But, parallel processing still bears the advantage of a decreased vulnerability to failures and slow peers.

Milestone Messages

Up to this point, we introduced everything needed for estimating query completeness. Based on the routing trace included in each reply, we can argue about the actual routing graph of a query and estimate the number of expected replies from that. But, at any time of processing, the accuracy of the estimation depends on the amount of information available. For late replies, e.g., resulting from highly loaded peers, this means that accurate information about the corresponding parts of the routing graph are available late as well. Moreover, unavailable peers will break the chain of processing and information about the corresponding path(s) in the routing graph will never be received. One idea to overcome this burden, i.e., to quickly

increase estimation quality even with only a few replies available, is to send information from each routing point in a separate message directly back to the initiator, while forwarded plan copies are still processed. We call this kind of message *Milestone Message*, MiMe for short. Note that these messages are not needed in general to make the method work, because they only include information that is available in the routing trace of a reply as well. Rather, they help to improve the accuracy of the estimation in early states.

In our approach, each peer corresponding to a routing point in the graph sends a MiMe to the initiator in order to tell him "This plan is still processed" and to provide some valuable information for completeness estimation. Note that MiMes should not be confused with heartbeats (or similar), which are sent in regular intervals. Rather, MiMes are sent irregularly every time a certain part of the query plan is processed – and they are not essential for applying the proposed completeness estimation. Among other small fixed-size parts, a MiMe contains a query ID, the actual routing level, and the number of forwarded query plans or range information. The only part that is not fixed in size is information about the overlay structure. These are $|\Pi|$ bits for each path Π that is unknown to the query initiator. We will discuss the background of this in Section 7.2.1.

Overhead MiMes introduce some communication overhead, as extra messages need to be sent through the network. Thus, whether to send MiMes or not should be decided on the basis of the physical query plan. Intuitively, the number of MiMes M_Q resulting from processing a query Q correlates with the number of routing points in the routing graph and thus, with the number of replies to Q. More exact, each peer on a routing level lower than the maximal level L sends such a message before it forwards any query plan. This is true for all routing points but those that correspond to subsequent peers (not the first one) processing a sequential operator. A peer starts the processing of a routing operator, determines the set of keys or key range that plan copies have to be routed to, and initializes this routing using the overlay functionality. Before this last step, a corresponding MiMe is sent to the query initiator. Each peer that receives such a plan copy processes the operator and all subsequent ones that can be processed locally. Afterwards, it acts the same starting the processing of the next routing operator. Note that M_Q can be seen as a measure of the degree of parallelization.

Assume $l(rp)$ represents the routing level that a routing point rp is located on. Generally, with respect to the routing graph $RG = (V, E)$,

$$M_Q = |\{v \in V | \exists u \in V : l(u) = l(v) + 1 \wedge (v, u) \in E\}| \ .$$

If we define RP_l as the set of all routing points rp on level l and $d(rp)$ as the out-degree of one routing point,

$$M_Q = \sum_{l=0}^{L-2} \sum_{rp \in RP_l} d(rp) \ .$$

Processing an operator op on routing level $0 < l < L$ using a range query (i.e., applying DOR) results in

$$d(rp) = |r_l|$$

MiMes, where $|r_l|$ represents the number of peers responsible for the range queried in op, e.g., all peers responsible for a certain attribute. Assuming a constant amount of data, $|r_l|$ increases with a growing number of peers N in the system and decreases with a growing replication factor. Thus, $d(rp) \in O(N)$ holds if the replication factor is regarded as a constant. If the total number of data items D in the system increases with N, this will result in a sub-linear asymptotic bound depending on the actually queried range. $|r_l|$ MiMes are sent for each routing

point on level $l-1$. Thus, the total number of MiMes on a DOR level depends on the degree of parallelization on levels $l-1$ and below. Processing a query Q using only range query operators results in

$$M_Q = \sum_{l=1}^{L-1} \prod_{i=1}^{l} |r_i| \quad \in O(N^{L-1}) \; .$$

For FNR follows

$$d(rp) = |kl_{rp}| = s_{rp} \cdot \frac{D}{N} \; ,$$

where $|kl_{rp}|$ represents the number of key lookups that are initiated at rp to process o. This number of generated query plans can be expressed using a selectivity s_{rp} in combination with the average amount of data a single peer is responsible for. Without loss of generality, we expect a perfect load balancing. $d(rp) \in O(\frac{D}{N})$ holds if load balancing is supported. Again, the replication factor is regarded as a constant. This is a rather theoretical bound, because in practice the internal optimizer will only choose to apply an FNR if $|kl_{rp}|$ is rather small. Assuming constant D, this becomes $O(1)$. This is intuitive, because data items are distributed among participating nodes as long as the amount of local data does not fall below a given lower bound. All the same how many peers participate, if D stays constant, the number of generated plan copies will not change, because FNR only depends on data level. If a query Q is processed using only FNR, this results in

$$M_Q = \sum_{l=0}^{L-1} \sum_{rp \in RP_l} s_{rp} \cdot \frac{D}{N} \; .$$

If D is constant, the number of key lookups initiated at one routing point will decrease with decreasing $\frac{D}{N}$. But, the number of lookups over all routing points of a routing level stays constant with growing N. Thus, $M_Q \in O(1)$. This underlines the fact that DORs become more expensive with growing N, but FNRs reveal constant costs of processing depending only on the actual set of data items. For a combination of both, DOR and FNR, the asymptotic notations from above have to be combined accordingly. The total overhead is bound by the more expensive one, i.e., $M_Q \in O(N^{L-1})$.

Neglecting constant parts of a MiMe like message header and similar, the size of one MiMe is

$$B_m = C + \sum_{\text{unknown paths } \Pi} |\Pi|$$

bits. C refers to the fixed-size part of a MiMe. As each part but the set of unknown paths is constant, this is in $O(N)$, regardless if D is constant or not. Respecting the fact that with proceeding time the number of paths unknown to the initiator will approach 0 (especially if caching is enabled), we may regard the bandwidth overhead as constant.

The benefits gained from using MiMes are reflected in an increased quality of the estimation, see Section 7.3. In Section 8.2.5, we additionally evaluate the effect of using MiMes on an empirical basis.

The Method as a Whole

Based on the definitions and ideas explained above, the whole process of completeness estimation is now illustrated using a simple example. Summarizing the method, the query initiator extracts information about routing points from the routing traces and optionally from MiMes. According to the routing method applied on each level, the initiator applies the corresponding

formula(s) for estimating query completeness and thus, is able to determine an (updated) minimal number of replies expected. This number is guaranteed to be minimal by assuming one reply for each unknown subtree of the routing graph. Intuitively, this results in a completeness value $\frac{r}{R}$ that is guaranteed to be always above the actual completeness. This is true because we assume that no plan copy is dropped. In Section 7.3, we discuss alternatives to this minimal estimation. As the minimal number of expected replies is adopted during processing as soon as new replies (more exactly: new information about the routing graph) are received, the estimated completeness will reach the actual completeness gradually. The more information we already gathered, the more accurate are our estimations. We will analyze this more detailed in Section 7.3.

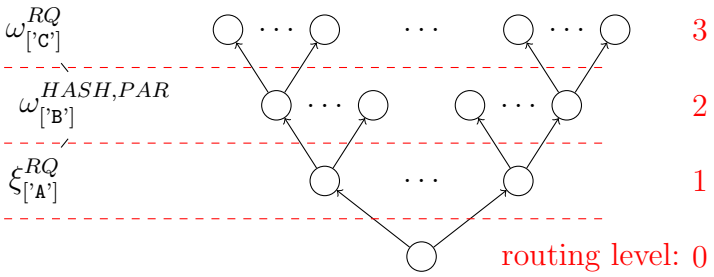


Figure 7.4: Example query

Imagine the query plan in Figure 7.4. $\xi^{RQ}_{['A']}$ extracts all data items for attribute A using a range query to contact all peers responsible for a part of A. Thus, first routing method is DOR. $\omega^{HASH,PAR}_{['B']}$ and $\omega^{RQ}_{['C']}$ materialize attributes B and C as additional information for each input data item. This means, second routing method is FNR, third is DOR.

At time t, the following knowledge is available at the query initiator:

- the ranges $r_{\mathtt{A}}$ and $r_{\mathtt{C}}$ of attributes A and C
- subsets $K_{\mathtt{A}}$ and $K_{\mathtt{C}}$ of all paths from peers responsible for a part of $r_{\mathtt{A}}$ and $r_{\mathtt{C}}$
- a subset D_1 of all peers on level 1 with their fanout d, $|D_1| = |K_{\mathtt{A}}|$

Naturally, we assume $t \geq t_0$, where t_0 is the time of the first reply. Before t_0, completeness is always 0. From $r_{\mathtt{A}}$ and $K_{\mathtt{A}}$ we can estimate a minimal number p_1 of peers on level 1, as shown before. Thus, the following holds:

$$p_2 = \sum_{d \in D_1} d + (p_1 - |D_1|) \quad .$$

This corresponds to the minimal number of replies to be expected, because we assume an outdegree of 1 for each peer on level 1 for which we do not have exact knowledge. In analogy to p_1, we estimate d_2 (the fanout of each routing point on level 2) from $r_{\mathtt{C}}$ and $K_{\mathtt{C}}$. Finally, the number of expected replies at time t is

$$R = p_3 = d_2 \cdot p_2 \quad .$$

This shows why assuming static query planning eases completeness estimation and improves its quality: we can assume the fanout of d_2 for each peer from level 2, even for those for which we did not receive any information yet. Adaptive query processing can result in heterogeneous subtrees

and routing levels. Thus, we cannot deduce from one subtree to another. For this, information for each vertex in a routing graph has to be maintained separately. Nevertheless, we can stick to the same general principle. Quality of the prediction is lowered and storage overhead slightly increased. Similar applies for the support of other processing strategies like inter-operator parallelism or spanning trees. For instance, inter-operator parallelism can result in different routing strategies on the same routing level. Spanning trees do not influence completeness estimation if according information is included into the routing traces (and MiMes).

Implementation of the proposed approach is rather easy and light-weight. We only have to keep small information for each routing level (namely the routing method, estimated number of peers on this level, queried ranges for DOR, and out-degrees for FNR). Dropped query plans will result in a routing graph with leafs on different levels. Each leaf beneath the level of replies represents a dropped plan copy. This can be integrated straightforward by sending a corresponding acknowledgment (instead of a reply) and processing this information accordingly. Further, we have to detect replicated replies, which can occur in replicating overlays. We use the term replicated replies, because they do not result from contacting the *same* peer multiple times. Rather, they result from contacting multiple *replicated* peers. This is done by the DHT when acknowledgment messages get lost or timeouts are too short. Then, many DHTs just resent the according message to a peer replicating the queried data. This can result in two peers processing the same plan copy, and thus, in replicated replies. Fortunately, replicated replies can be identified from the routing trace using IDs of each routing point. Further, the DHT has to provide estimation techniques for every DOR method used.

Overhead The proposed method comes along with a very small overhead. In terms of messages there is no overhead at all, because any information needed is included into query plan messages and reply messages. This is true as long as MiMes are not used. That overhead is discussed above. The bandwidth consumption resulting from applying completeness estimation is almost negligible. Query plans contain only small additional information, which is the routing trace and a sketch of the overlay structure as it is known at the query initiator. The size of the routing trace depends on the size of the routing graph (more exact, on the maximal routing level L). Each entry of the routing trace corresponds in size with one MiMe. The overlay structure grows with the number of peers participating in the system. In P-Grid, we have $|\Pi|$ bits for each path Π known to the current peer. Thus, the bandwidth overhead for each plan copy generated at routing point rp is in bits:

$$B_{rp} = \sum_{\text{known paths } \Pi} |\Pi| + B_m \quad \in O(N) \quad ,$$

where B_m refers to the size of the MiMe sent from this routing point. Note that information of size B_m is always added to a query plan, regardless if MiMes are sent separately or not. Thus, bandwidth consumption for one query neglecting MiMes is

$$\sum_{rp \in RG} d(rp) \cdot B_{rp} \quad \in O(N \cdot N^{L-1}) = O(N^L) \quad .$$

Further Aspects

Some further issues of estimating query completeness for the proposed framework of query processing have not been discussed so far. In the following, we briefly summarize them.

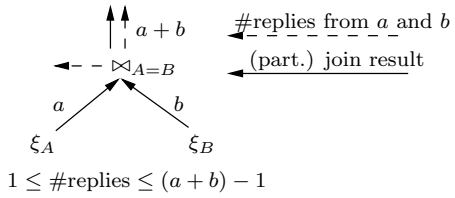


Figure 7.5: Completeness for parallel join

Inter-Operator Parallelism Until now, we only considered sequential and intra-operator parallelism. This implies that a query plan is always processed in strict post order. For binary operators, execution of both input sides in parallel proved to be very efficient. Completeness estimation in this case is straightforward. Technically, we now have to handle parallel routing levels and apply according estimation formulas at the routing points combining different branches of a query plan. Imagine the join in Figure 7.5. The number of plans generated by the leaf operators are estimated using the methods explained. If we estimate a plans from the left extract operator and b from the right side, the join will produce $1 \leq r \leq a + b - 1$ replies or forwarded query plans. Note, this holds for all join operators. Duplicated replies are prevented by using one dynamically chosen join peer for a parallel join operator. $a + b - 1$ replies are received if input from both sides is received in an alternating manner and results are transmitted immediately. If the join peer intermediately caches data or receives multiple plans from one side in sequence, the number of replies shrinks. The according information is simply added to the reply or MiMes and completeness estimation can be adjusted.

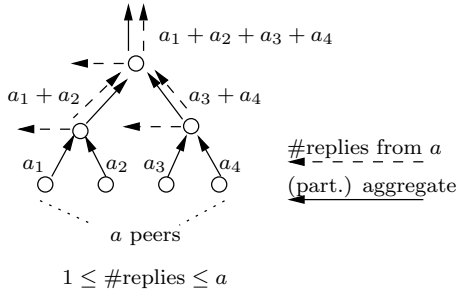


Figure 7.6: Completeness for parallel aggregate

Other Advanced Processing Techniques We designed the presented method to be extensible by a wide variety of advanced processing techniques. This involves inter-operator parallelism like different join strategies and sophisticated aggregation techniques. The techniques need to be classified according to the introduced routing methods, and the principles of other operators from the same class need to be applied. As a last example, imagine an aggregation technique based on spanning trees (e.g., [121]) as illustrated in Figure 7.6. Peers on leaf level of the spanning tree extract data. Afterwards, partial aggregates are determined and forwarded until a final aggregate is delivered. Similarly to the inter-operator parallel join, the number of replies to expect is $1 \leq r \leq a$, if a peers are on leaf level. The initiator receives a replies if the spanning tree has a height of one and himself is at the root. It receives one reply if another peer is on top of a spanning tree. If partial aggregates are tolerated (which is a

good choice in the concerned systems), the number of received replies may vary between these extremes. Note that for this issue, applying completeness estimation at the peers one level above the leafs can help to decide when to deliver partial aggregates. All the same, whether the spanning tree is built dynamically or not, completeness can be estimated satisfyingly when integrating the needed information in replies and heartbeats.

Adaptive Query Processing In the considerations above, we always assumed that all logical operators of a query plan are replaced by physical ones at the initiating side. Especially in dynamic settings, an adaptive approach should be preferred. This can be achieved by replacing only some logical operators at a peer and retaining logical operators in the query plans. At runtime, these logical operators are replaced successively and their physical counterpart is executed. For completeness estimation, logical operators are consequently handled as dynamic routing methods. Even if, for instance, a queried attribute is known, the actual involved part of the overlay is not known due to the variety of possible access techniques and indexes. The dynamic transformation of the query plan reflects in *(i)* more unknown parts in routing graphs, and *(ii)* in different routing methods at the same level of this graph. In the static case, with the information contained in the first reply received, we know about all applied routing methods because they do not differ between routing points of the same level. As introduced, we expect a minimal value of one for the number of replies from each such operator. Again, replacing this by an average value or similar is by all means conceivable. The idea of caching can help here too.

7.2.1 Overlay-Intern Completeness Estimation

FNRs bear a significant advantage for completeness estimation: each routing trace or MiMe that includes the routing point initiating the FNR also contains the out-degree of that point. Thus, the number of expected replies can be determined exactly as soon as the first of these plans is replied. A harder problem is to estimate completeness for DORs, because the resulting number of follow-up plans is not known before initiating the routing. But, DOR is used in a multitude of physical operators. We restrict ourselves to the usage of range queries. A concrete method for estimating the number of generated plans is due to the used overlay. Approaches for this involve querying liveliness of the peers before sending any queries to them in order to detect and maintain faulty routes. In some cases, statistical peer availability estimations are also used. The statistical estimation methods like [37, 215] are useful for estimating the future availability of peers that already participated in the overlay. The probing techniques like [137] are useful when active measurements are needed at the cost of communication overhead.

A method for estimating the completeness of range queries in P-Grid and its applicability to other overlay systems is proposed and discussed in [109][1]. With this approach, we are able to estimate query completeness for all proposed physical operators. We summarize this approach in the following.

Estimating Completeness of Range Queries in P-Grid

The way P-Grid handles range queries is briefly introduced in Section 4.1.1. The most efficient approach is based on the shower algorithm. Though it is guaranteed by the shower algorithm that all peers receive exactly one range query message, it is currently not possible for the initiating peer to estimate the number of peers concerned by a range query, i.e., estimating

[1] A work emerged from a collaboration with the people from EPFL, who designed the P-Grid overlay system.

the number of response messages it has to expect. For keyword based queries, a peer receives only one query response by one peer in a structured overlay network, because only one peer (or any of its replicas) is responsible for the given keyword. A peer is therefore able to determine when a query finished and when it received all matching items to either inform a user, start post-processing, or initiate subsequent queries. This is currently not possible for range queries in structured overlay networks as the number of response messages depends on the number of peers in the target range, which is usually not known to a peer. We propose to estimate this number based on the local information available in a peer's routing table. Errors are corrected by intermediate peers forwarding range queries and by peers responding to range queries. We thereby assume a load-balanced system, in which each peer holds approximately the same amount of data. Load balancing in DHTs has been studied intensively, because it is one of the main principles of DHTs. [7] presents its realization for P-Grid, including an evaluation on a real-world test-bed.

Basic Approach for the Shower Algorithm A peer initiating a range query begins with determining the interval bounds of the desired range. Afterwards, the intermediate peers responsible for routing the plan message forward it to one or more subtrees, depending on the peers' local paths, the paths of peers from their routing table, and the paths of the queried range. The crucial point is to estimate the number of peers responsible for a certain key range. But, due to load-balancing aspects, this is quite difficult. Parts of the P-Grid tree can be arbitrarily deep. The more index items are mapped to one key range, the deeper the subtree for this range gets in order to achieve load balancing. Thus, we cannot deduce the size of a range based on knowledge about other ranges. The idea is to use all available path information in order to build an estimated P-Grid tree. Based on this tree, we can determine an expected minimal number of replies.

Next, we explain how we can determine the minimal number of replies from an estimated P-Grid tree. Let

$$b_1 b_2 b_3 \dots b_x$$

denote the x bits of the path of a peer. From this path, we can deduce the existence of at least x other peers: Let $\overline{b}_i$ denote the inverted bit b_i. For each path

1: $\overline{b}_1$

2: $b_1 \overline{b}_2$

3: $b_1 b_2 \overline{b}_3$

...

x: $b_1 b_2 \dots b_{x-1} \overline{b}_x$

there must exist *at least one* responsible peer. Knowing about several paths from peers in a range, the initiator can deduce a minimal number of peers in that range. In order to achieve this, the initiator builds a tree from those paths and reflects to the minimal number of peers.

Figure 7.7 illustrates this. The figure shows an example P-Grid tree. Assume a query for the range $00100 - 1101$ was initiated. Further, the initiator P_0 knows about four peers, where the paths from P_1, P_2, and P_3 are in the range. Every peer stores at least one reference to another peer for each of the bits of its path. Thus, P_0 has at least knowledge about four peers, each located in a different subtree. In the figure, the part of the tree that the initiator can determine from its local routing information is shown in solid lines. The dashed lines indicate the part of the tree that is not known to the initiator, which results in a small error in this first estimate. The minimal number of peers in the range estimated in this situation is 8, whereby the correct value is 10.

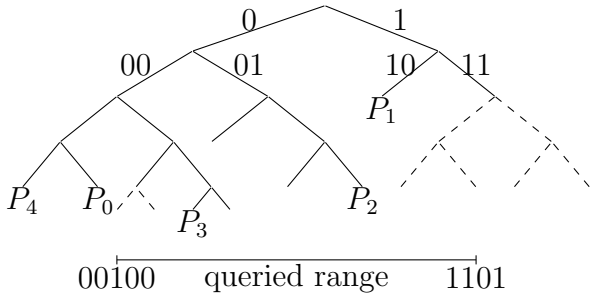


Figure 7.7: Estimating the P-Grid trie

Estimation Refinement The first estimation performed by the query initiating peer is solely based on the routing information available at that peer. The local routing table stores at least one reference per level. In other words, we know about at least one peer of each subtree that a range query is sent to. For achieving fault tolerance and load balancing, structured overlays usually keep multiple references at each level. This allows to remain operational during peer churn and to select the least loaded peer for query load-balancing. Therefore, the information available for the initial estimation increases with the number of references per level.

But, the information gathered like this is still not complete. The estimated value might still be too small, because some peers remain "invisible" from the local point of view. Therefore, initiating peers piggy-back the estimated number of peers in a range with each initiated range query. For example, in Figure 7.7, the range query sent from P_0 to P_3 also contains the estimate that three peers are contained in the subtree 001*. P_0 only knows that there has to be a peer 00111 and at least one peer for 0010. P_0 does not know that the subtree 0010* actually consists of two peers. P_3 is aware of this fact, because P_3's routing table contains at least one of the peers from subtree 0010*. Thus, it can include the correct number of peers in the forwarded plan copy. After receiving the reply, P_0 can correct the estimate for the queried range. Using MiMes, this can be done even before receiving this reply. Peers receiving a range query with a correct estimate do not have to include any corrected information. We extend this to not only store and ship the estimated values, but also the paths known at the initiator, the intermediate peers, respectively. Actually, the estimated value can be determined from the paths included. Like this, we result in slightly more bandwidth usage, but we can correct the whole estimated tree at the query initiator rather than only the estimated value for the current query.

The required message overhead is rather small, because no additional messages have to be sent and only small information is piggy-backed with queries and replies. In case a query plan containing incorrect information hits a peer outside the target range, the receiving peer has two choices. Either it can react by replying with a short acknowledgment message correcting the initial estimate, or it forwards the incorrect estimate to target peers in the range and relies on later corrections. In the first case, the query initiator can sooner correct the estimated completeness at the cost of a small extra message. In the second case, the correction is done at a later time with the reception of query results, but without additional messages.

Applying the method as described above, we will never over-estimate the number of expected replies. Moreover, when a query is finished, we will always recognize this for sure. This is possible because the paths of the replying peers are analyzed. Thus, when all replies are received, we know that each path in the subtree is covered.

Tree Caching There was much research spent on designing overlay systems as much stable and reliable as possible. This means, the topology of the DHT (in our case the P-Grid tree) should change infrequently and slowly, even under churn. Thus, we might cache estimated trees once they are built. These cached trees can be used later for subsequent queries. Feedback from replies can be used to adapt trees to changes in the overlay structure – which may, of course, occur, but are expected to be rather rare. In this way, we achieve a quite accurate and satisfyingly exact completeness estimation from the beginning of a query, which is automatically maintained with each initiated query.

Caching the estimated P-Grid tree can be useful for several other tasks as well. For instance, information about the overlay can be used to reduce message overhead in parallel operators, e.g., by identifying peers that are responsible for a set of q-grams. The corresponding issues are highlighted in Section 6.1, where we compare sequential hash-based operators with parallel ones. It can further be used to decide about the dimension to which local skyline queries are sent in Φ^{PAR} and Φ^{HYB}. Note that caching also information about data distribution can help to improve the estimation for FNRs. Caching introduces the problems of consistency and cache maintenance, which are not in the focus of this paper.

Another interesting problem in this context is when to discard paths in the cached tree. This question is particularly interesting, because inconsistencies in the set of paths may also occur during the processing of a single query. For instance, during a couple of tests in the very unreliable PlanetLab test-bed [50], we experienced peers responsible for a path like 001 and others responsible for paths like 0010 and 0011. Usually, this should not happen. Since this is an issue of the underlying DHT, we do not further discuss it and expect consistent P-Grid paths.

Usability in other Overlay Systems Our approach is based on a parallel resolution of range queries in a binary trie similar to a prefix hash tree. In the case of P-Grid, the depth of each subtree can be estimated by the nodes from this subtree stored in the local routing table. To the best of our knowledge, no other DHT system can already provide completeness estimation for range queries. In the following, we briefly discuss the possibilities for other systems to estimate the number of query replies based on our approach.

The approach for range queries in SkipGraphs [17, 16] is the one most similar to the one of P-Grid. Peers also maintain routing information at multiple levels. Thus, the proposed method can be used by SkipGraphs to estimate the number of peers in certain subtrees. The only problem is the number of peers remaining in the bucket layer below the lowest interconnected skip-list level. But, as load balancing is in place, this number should be similar to the number of buckets that the current node belongs to.

Approaches like [167] and [130] are based on a prefix hash tree. In such a tree, peers remain at each level of the tree, in difference to P-Grid, where peers only remain at the leaf level. The routing in a prefix hash tree starts at the root level and trickles down the tree from nodes to their children, until all nodes in the target range are reached. As we assume that nodes do not know the exact number of their children, it is not possible for them to estimate how many nodes will return results for a range query. If this number could be estimated, the technique presented here could also be adapted in these systems.

Finally, approaches forwarding a range query sequentially along neighbors cannot estimate the final number of nodes involved in a range query. This holds for CAN-based systems presented in [80, 182] and the CHORD-based system Mercury [29]. But, as explained above, completeness estimation for sequential operators is nevertheless easy, because only one single query plan is forwarded.

7.3 Guarantees

Completeness estimation as introduced up to this point is a nice tool, but it gains really significance and importance only if we provide guarantees, more generally, meaningful measures for the quality of estimation. This section deals with this important aspect. First, we have to consider which kind of measurements are conceivable and meaningful in order to weight the accuracy of the estimated query completeness. We distinguish between three different kinds of guarantees:

1. A general guarantee for the accuracy that says: "If completeness is predicted as $c\%$, this is true with a probability of $x\%$".
2. A guaranteed boundary: "If completeness is predicted as $c\%$, this is above/below the actual completeness with a probability of x%".
3. A confidence interval: "If completeness is predicted as $c\%$, the actual completeness is between $(c-y)\%$ and $(c+z)\%$ (with a probability of $x\%$)".

Which guarantee is preferred will depend on the specific application. A general guarantee as in point 1 may be very helpful, but in many situations a guaranteed lower bound will be more interesting (i.e., $x = 100\%$ for the lower bound in point 2). This applies, for instance, for aggregation queries aiming at a specific degree of accuracy. A confidence interval as in point 3, if small enough, may be the best tool to correctly judge an estimated value for query completeness from a standard users point of view.

We start with proving a fundamental characteristic of the proposed approach: the estimated query completeness is 100% iff the result is finally complete. This means, at the point in time when we estimate that 100% of all replies are received, there will no results arrive later for sure. In the other direction, when all replies are received, the estimated value for query completeness is guaranteed to be 100%. If we estimate a completeness of 100%, there will be no uncertainty at all about this value. Note that this does not only hold for peer level. If all replies are received, then all data will be received as well. Thus, this guarantee holds also for data level.

In fact, by this we can provide more than the most other P2P information systems. But, a crucial point is the probability of achieving actually complete results. This depends on the reliability and dynamics of the system – and on the DHT handling these issues. Complete results should not be expected in general. We illustrate this in Figure 7.1 on page 149. Since most of the run queries do not result in really complete results, we achieve an estimated value of 100% only occasionally. Thus, with the basic characteristic, we can provide a guarantee in the fewest cases. Moreover, guarantees for situations before achieving complete results are much more important. We approach this by first relaxing the requirements to meet the 100% guarantee: it holds from that point of time when we achieve *minimal routing graphs*. Details are presented subsequent to the proof of final correctness. Further, to provide guarantees even before achieving minimal routing graphs, we introduce three different methods to estimate missing parts of routing graphs. Based on these methods, we extend the approach by probabilistic guarantees. Actually, these are the only guarantees we can provide below an estimated query completeness of 100%. Next, we proof that a determined completeness of 100% is guaranteed to be correct.

Proof of Final Correctness

Let C be the estimated value for query completeness. The following theorem captures the 100% guarantee introduced informally in the last section.

Theorem 7.3.1. $C = 1.0 \Leftrightarrow$ *all replies are received.*

Proof of Theorem 7.3.1. **Direction** $\Rightarrow$: We prove this direction of the theorem by induction over the number of routing levels L. This means, we have to prove that if $C = 1.0$, no reply can arrive later, i.e., the final number of leafs in the routing graph cannot be underestimated.

1. Basis:
   - $L = 0$: The query is completely processed locally, no routing at all is initiated. In this case, we receive exactly one reply, which corresponds to the one and only leaf in the routing graph. The only routing point in the graph corresponds to the query initiator. Note that this is not true if the initiator "routes" one or more messages only to itself, i.e., to a key or set of keys itself is responsible for. In this case, there are multiple routing levels in the graph and each routing point corresponds to the initiating peer.
   - $L = 1$: Exactly one routing is initiated by the query initiator. The applied routing method is either *DOR* or *FNR*. We use a case differentiation:
     (i) FNR: Using FNR, we exactly know the out-degree of the one routing point on level 0, because at least one reply is received. Thus, the number of leafs in the graph is guaranteed to be correct.
     (ii) DOR: Using DOR, we know the addressed key space r (prefix of queried paths, i.e., the queried range) and k paths $p_1 \dots p_k$ from responsible peers, $k \geq 1$. We know this, because at least one reply is received. k corresponds to the number of replies already received. Again, we have to distinguish two cases:
     (α) $p_1 \subseteq r \rightarrow k = 1 \rightarrow$ only one reply is sent $\rightarrow$ the number of leafs (that is, 1) is correct
     (β) $r \subseteq p_i \forall i \leq k \rightarrow$ there is *always* the guarantee that we recognize if one or more replies are missing $\rightarrow$the number of leafs is correct
     The proof for this is straightforward and omitted here. It can be deduced directly from the approach for estimating range queries introduced in Section 7.2.1. The idea is to use an induction over the path lengths.
2. Hypothesis: the number of leafs is estimated correctly for $L = l$
3. Inductive step: the number of leafs is estimated correctly for $L = l + 1$
   - From the inductive hypothesis we can deduce that the number of routing points rp_i estimated for routing level l is correct. Thus, we have to prove that the out-degree for each of the routing points rp_i is estimated correctly. This is done by a case differentiation:
     (i) If we have at least one reply from the subtree rooted by rp_i, the routing trace from that reply will be known. Thus, we know for sure whether one or more replies are missing. More details on this can be found in the proof of the opposite direction.
     (ii) If we have no reply from that subtree until now, we will estimate one reply as the minimum from this subtree. This contradicts to $C = 1.0$

Direction $\Leftarrow$ This direction is proven on the basis of an indirect proof. We have to show that the following is a contradiction: If actually all replies are returned, we might still expect a reply from one or more subtrees in the estimated routing graph (i.e., $C < 1.0$). This can be true only if we expect a subtree in the routing graph that actually does not exist, i.e., we over-estimated the out-degree of at least one routing point.

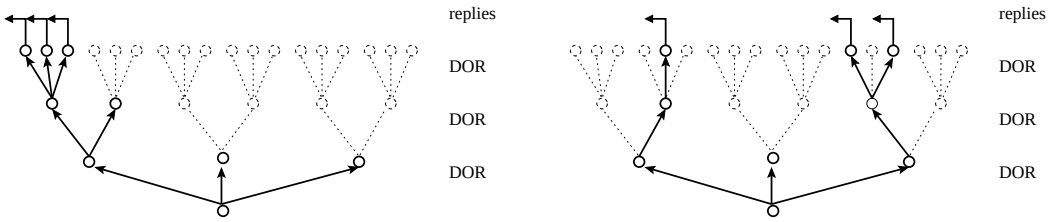
(i) In the case of FNR, this contradicts to the information gained from the routing trace: if the out-degree of a routing point is known to be d, then there have to be exactly d subtrees (thus, $\geq d$ replies).

(ii) In the case of DOR, this contradicts to the prove of the other direction: by rebuilding the P-Grid tree we are sure about whether a range is completely processed or not. We already explained this in Section 7.2.1: each path in the queried range has to be covered by one replying peer – only when all these peers replied the range is expected as completely processed. Thus, we will never over-estimate the number of subtrees. □

Extending the proof to heterogeneous subtrees in the routing graph (due to adaptive query processing) is straightforward.

Probabilistic Guarantees

From the technical side, providing a confidence interval with a significant accuracy is very challenging. This is due to the unpredictable nature of the underlying systems and the potentially irregular structure of the used DHT overlay. One approach to handle this is to introduce probabilistic guarantees. As explained, we always predict an expected number of replies that is guaranteed to be below the actual number of replies. This results in a predicted completeness $\frac{r}{R}$ that is guaranteed to be above the actual query completeness. We can prove this similar to the proof of Theorem 7.3.1. We omit the details here. Note that this guarantee holds for peer and also for data level.

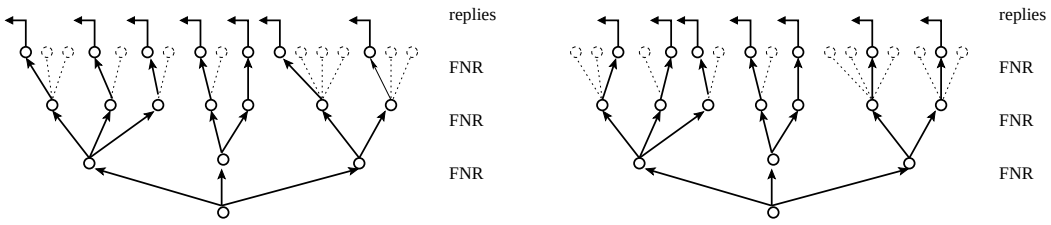


(a) left-bound (b) random

Figure 7.8: Minimal routing trees for complete DOR

Before we get over to probabilistic guarantees, we relax the conditions needed to achieve a completeness estimation guaranteed by 100%. In fact, we already reach this point as soon as all needed information about the routing graph is achieved. This may already be the case with relatively few results, depending on the applied routing methods and the actually received replies and MiMes. For DOR, we need to know about the queried range (known when planning the query) and the exact number of peers in that range. This is achieved when one of all identical subqueries is completely answered, all the same from which part of the routing graph. This can be true also if no subtree is completely known at all, depending on the paths of replying peers. If all paths of a range are known, the exact number of responsible peers is known. Figure 7.8 illustrates this. Here, we introduce the notion of *minimal routing graphs*. All routing points and connections drawn solid are needed at minimum in order to achieve the 100% guarantee. We picture two out of many possibilities. The first one shows the needed information "as much left-oriented as possible", the second one is random.

The same is illustrated for a routing graph containing only FNR levels in Figure 7.9. The mixture of both is omitted, because it is intuitive. In the FNR case, much more information is



(a) left-bound (b) random

Figure 7.9: Minimal routing trees for complete FNR

needed in order to achieve an exact completeness. The out-degree of each routing point must be known. Thus, at least one result from each routing point on level $L-1$ must be received. This implies complete knowledge of the routing graph below level L.

The notion of minimal routing graphs helps to understand the possible errors we make during completeness estimation. If any of the solid parts in Figures 7.8 and 7.9 is missing, we potentially include such an error. The depth of the routing graph is known as long as we expect homogeneous subtrees, because at least one reply is already received. Thus, the only errors we can make are:

- estimating a wrong number of peers in a range when applying DOR
- estimating a wrong out-degree of a routing point when applying FNR

In order to provide a probabilistic guarantee, we have to weight these potential errors for each point in time when we estimate the query completeness C as $0 < C < 100$.

Providing an estimated completeness that is guaranteed to be above the actual one is more than many systems can provide. But, of course, we are interested in a general accuracy and in a guaranteed lower bound. We investigate three different estimation techniques: estimate *(i)* a *minimal* number of replies, as introduced before, *(ii)* an *average* number of replies, and *(iii)* a *maximal* number of replies. Unfortunately, the mentioned unpredictable nature of the underlying systems prevents from providing exact guarantees for the latter two methods. After briefly introducing them, we will discuss how to provide probabilistic guarantees instead.

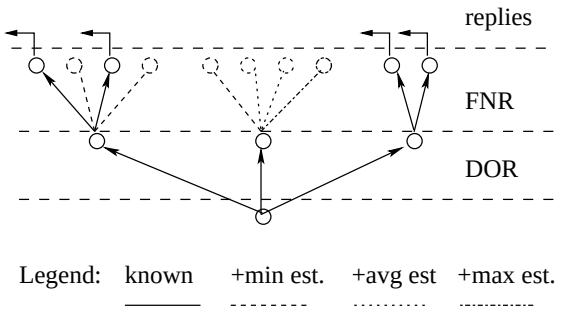


Figure 7.10: FNR example for minimal, maximal, and average estimation technique

All three techniques are based on the already received information about the routing graph. The idea is illustrated best using the small example graph in Figure 7.10. The middle routing point from level 1 is known, but not its out-degree in the FNR level. From the other two subtrees, we

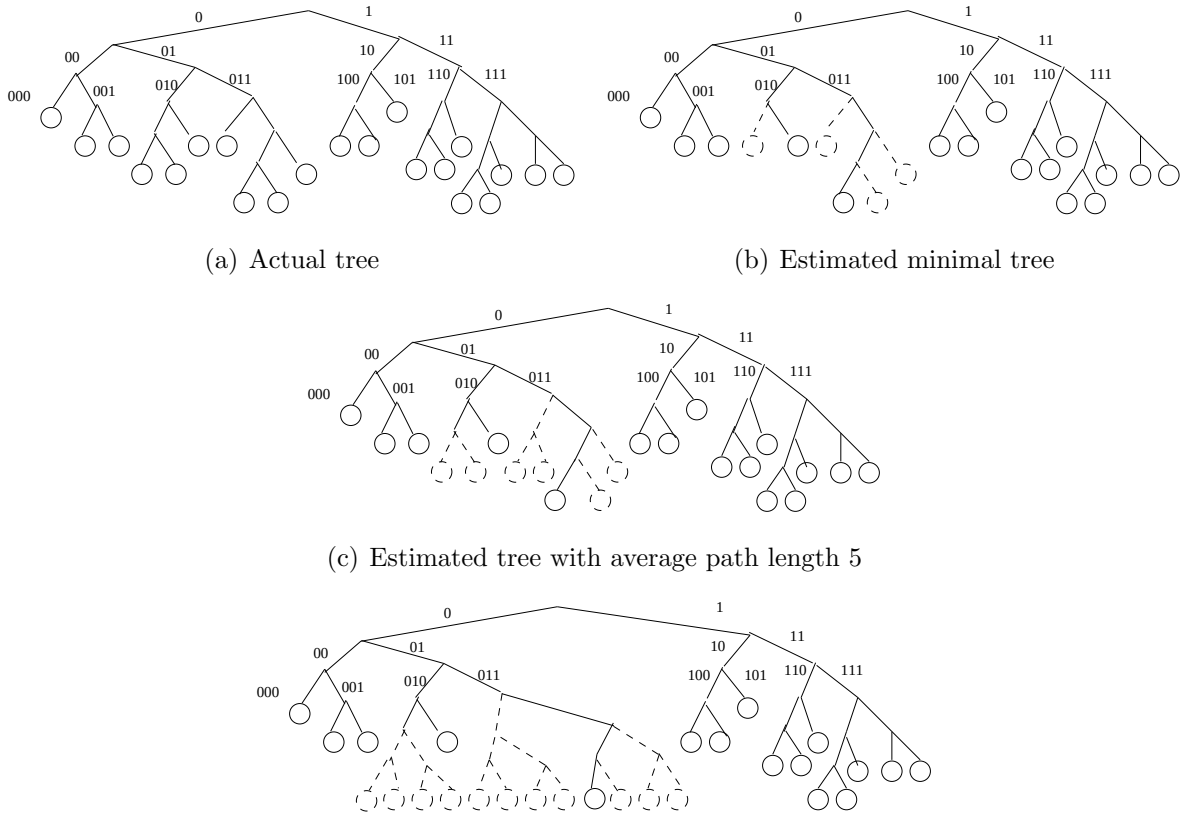
already received 4 replies. By this, we know that there will be 2 more replies for sure (we know the out-degrees of both routing points) plus at least 1 from the "missing" subtree. Thus, with the minimal method we estimate 7 replies. As its name suggests, the average method calculates the average of out-degrees on the routing level in question, which is 3. Thus, 9 replies are estimated. Similar, the maximal method uses the maximum of out-degrees and estimates 10 replies. The only guaranteed bound is provided by the minimal method. The actual number of replies can be anything from 7 to a huge. This means, the actual number of replies may still be higher than predicted with the maximal method. We also tried to use a method based on maximal values learned during time and from different setups. We soon observed that this method is always far below the maximal method introduced above – and, as we will show in Section 8.2.5, this one itself is below the actual completeness in most cases and times. Thus, we focus on the three methods introduced so far.

The described methodology is applied for each routing point with an unknown out-degree. Average and maximum are always calculated on the already gathered information in the same routing level. According DOR estimations are based on the locally estimated P-Grid tree. As introduced, a corresponding tree structure is maintained at each peer. For the two new techniques, average and maximal path lengths from that tree are used to estimate the actual shape of the P-Grid tree. Conceptually, each path in the range that has to be covered by one or more peers that we do not know yet is replaced by according paths with average and maximal length. The resulting tree is used to estimate the number of peers in a specific range. Note that caching the estimated tree can improve the accuracy of this steadily.
We illustrate this again on the basis of an example. Imagine a situation similar to that in Figure 7.10, but with a second DOR (range query for prefix 01) instead of the FNR. We start with an estimated P-Grid tree built on local knowledge. Assuming that two replies are already received, one from path 0101 and one from 011100, the three different techniques produce the following estimates:

- minimal: we estimate 18 replies, because there are at least 6 peers in the range, see the estimated tree in Figure 7.11(b)
- average: we estimate 24 replies, because the average length over all known paths is 5, see the estimated tree in Figure 7.11(c)
- maximal: we estimate 39 replies, because the maximal length over all known paths is 6, see the estimated tree in Figure 7.11(d)

The actual number of replies in the example is 21, because there are 7 peers in the range (see Figure 7.11(a)). Generally, the average and maximal techniques may still both determine values below as well as above the actual number of replies. But, the maximal method can be expected to be below the actual completeness. Here, we could also use maximal path lengths learned from history.

To increase the significance of all three methods, we now explain how to combine them with probabilistic guarantees. We can provide probabilistic guarantees for all values of $C < 1.0$. The guarantees described next also capture the case for $C = 1.0$, i.e., they become 100% in this case. We assume the out-degrees of routing points, the path lengths of nodes in the overlay tree, respectively, to follow a specific distribution model. In principle, any distribution model can be applied, even histograms for describing arbitrary distributions can be used. Problematic is the complexity of the following calculations. They involve determining the convolution of individual distribution functions which mostly depends on solving complex integrals. [202] gives suggestions and instructions for some of the most popular models. We refer to the Poisson distribution for out-degrees and path lengths for two main reasons. First, we observed that



(a) Actual tree

(b) Estimated minimal tree

(c) Estimated tree with average path length 5

(d) Estimated tree with maximal path length 6

Figure 7.11: DOR estimation after two replies from prefix 01

this discrete model with its non-symmetric distribution function fits the actual distributions particularly good. We only have to shift values accordingly in order to represent non-zero values. Second, calculations based on this model are very easy and efficient to implement. The Poisson distribution has the comfortable characteristic that determining the convolution of n $\lambda_1, \ldots, \lambda_n$ Poisson distributed random variables can be calculated using a Poisson distribution with parameter $\lambda = \lambda_1 + \cdots + \lambda_n$. Last but not least, this choice eases the understanding of the following considerations. Nevertheless, guarantees can be determined on other distribution models as well. Though the Poisson distribution model eases calculations, the resulting recursive formulas can still get quite complex.

In order to provide a guaranteed lower bound for query completeness, we next present a formula for determining the probability that the estimated completeness is below the actual one. We expect all occurring random variables to be independent from each other. Let N_l denote the random variable that estimates the number of routing points on routing level l and n_l the actual number. Looking only on the maximal routing level, the probability $P(N_L = n_L)$ under the condition that n_{L-1} routing points exist on level $L-1$ is

$$P(N_L = n_L | N_{L-1} = n_{L-1}) = \sum_{S_{L-1}} \prod_{i=1}^{n_{L-1}} P(D_i = d_i) \quad ,$$

where S_{L-1} is the set of all possible combinations $\{d_1, \ldots, d_{n_{L-1}}\}$ such that $d_1 + \cdots + d_{n_{L-1}} = n_L$. D_i describes the out-degree of an arbitrary routing point on level $L-1$. This formula computes the convolution of the n_{L-1} random variables. As mentioned above, using Poisson distributions

$$P(D_i = d) = \frac{\lambda_i^d}{d!} e^{-\lambda_i}$$

we can easily calculate this convolution using

$$P(D = d) = \frac{(\sum_i \lambda_i)^d}{d!} e^{-(\sum_i \lambda_i)} \quad .$$

Known out-degrees are simply excluded from this summation. This applies directly for distributions describing the observed out-degrees on FNR routing levels. A distribution model describing the path lengths in the overlay has to be mapped to a distribution model describing out-degrees accordingly. Currently, we do this using a repository of trees observed over time, but other approaches should investigated as well. Using the cumulative distribution, the current probabilistic guarantee is determined by

$$P(\text{estimated compl.} \leq \text{actual compl.}) = P(\rho \leq R) = P(D \leq R) \quad .$$

Including lower levels results in a recursive formula. Let K_l refer to the set of out-degrees already known on level l, k_l to the sum of all out-degrees in K_l. Then,

$$P(N_L \leq n_L) = \sum_{n_{L-1}=k_{L-1}}^{u_{L-1}} P(N_L \leq n_L | N_{L-1} = n_{L-1}) \cdot P(N_{L-1} = n_{L-1}) \quad ,$$

if we use $u_{L-1} := k_{L-1} + n_L - k_L - (k_{L-1} - |K_L|)$ for the maximal number of nodes that is possible on level $L-1$ (respecting known out-degrees and the estimated number of nodes on level L). Replacing $P(N_{L-1} = n_{L-1})$, this gets

$$\begin{aligned} &P(N_L \leq n_L) = \\ &\sum_{n_{L-1}=k_{L-1}}^{u_{L-1}} P(N_L \leq n_L | N_{L-1} = n_{L-1}) \cdot \\ &\sum_{n_{L-2}=k_{L-2}}^{u_{L-2}} P(N_{L-1} = n_{L-1} | N_{L-2} = n_{L-2}) \cdot \cdots \cdot \\ &\sum_{n_1=k_l}^{u_1} P(N_2 = n_2 | N_1 = n_1) \cdot P(N_1 = n_1) \quad . \end{aligned}$$

For levels of the routing graph that are completely known, corresponding probabilities resolve to either 1.0 or 0.0 and these parts could be extracted from the calculation. This is what we did with level 0 in the above formula. If individual parts of one level are unknown, this requires computing the convolution in the according routing level. With Poisson distributions and the information gathered for completeness estimation, this probabilistic guarantee can be calculated with ease. An according line $P = P(\rho \leq R)$ is added between lines 7 and 8 in Algorithm 17 on page 153. The determined probability P is returned together with the estimated completeness.

Note that these formulas also capture the point of time when achieving a 100% guarantee mentioned above. In this case, each involved probability is 1.0. Analog calculations can be used to determine a probability that the estimated completeness is above the actual completeness.

Thus, we can provide also probabilistic intervals. Of course, the significance of the calculations strongly depends on the quality of the input, namely the set of known out-degrees and path lengths, and the adequateness of the used distribution model(s). In early query states, the significance will be rather poor. This improves quickly with more received results. Including this fact into the determined probability dramatically increases complexity of the calculations. Evaluating different distribution models and the significance of applying them is left for future work.

7.4 Related Work

Existing works on completeness estimation exist in the areas of both, relational and distributed database systems. The concept of partial completeness in relational databases by means of *completeness constraints* is formalized in [141]. In [65], Devanbu et al. discuss the verification of query result completeness in relational databases. They expose the tuples directly neighboring to the left and right boundaries of the query result to the user for inspection. Further, they prove to the user that all the result tuples (including the two boundary tuples) are contiguous in the database. In [127], the notion of completeness is presented in terms of the relations that are true in the real world and the actual relations available in the database. It looks at the problem of answering queries on probably incomplete databases from the view of independence between queries and updates. In [146], the concepts of coverage and density are introduced in order to measure the completeness of databases at the intensional and extensional level. These concepts are used to determine the completeness of a database under a merge operator. [179] presents a query answering model for PDMS with the aim to achieve optimal completeness within given time constraints. In [37], a statistical approach for predicting machine availability in desktop grid and P2P systems is presented. A random walk method for counting and sampling peers in an overlay network is presented in [137]. All of the above works apply completeness estimation from a different point of view than we do. The proposed application scenarios are usually closely related to IR and focus on *general* data quality. We focus on completeness as the fraction of received results and expected results, which rather refers to a *current* result quality.

A very similar notion of completeness is used for dealing with aggregation queries in [26]. The authors define different levels of validity, which actually correspond to different degrees of completeness in the aggregate computation. They also suggest an algorithm to achieve validity, i.e., complete results. There is no discussion on how to estimate actual query completeness. But, the idea of validity classes, if combined with according estimation techniques, is very promising for transparent and user-friendly Public Data Management in dynamic environments. [103] defines *correctness* as the combination of *completeness* and *soundness*. Our point of view on query completeness is also different to these works. Completeness as they regard it should rather be referred as availability.

A recent system dealing with the problem of completeness estimation in structured overlays is Seaweed [145], which facilitates to estimate query completeness directly on data level. For this, Seaweed uses regularly updated data summaries representing the data distribution of neighboring peers. A heartbeat mechanism provided by the underlying DHT MSPastry [45] is utilized for maintaining the statistical information. In comparison to Seaweed, our approach targets different application scenarios and is based on different assumptions. We assume a wider variety of different query types and query processing strategies, exceeding the idea of broadcast and spanning trees. Our method for completeness estimation is designed to support more modern query mechanisms, such as similarity and skyline queries. The idea behind Seaweed is to provide delay-aware querying, which means data summaries and availability models of

currently unavailable nodes are used to predict query completeness and response times. Query processing is delayed until these nodes get available again. This does not conform to the in-situ querying that we propose. We suggest to let the DHT handle unavailable and new nodes, which is usually achieved by corresponding replication and applying routing techniques. We primarily use completeness estimation to measure the currently received fraction of a query result in the context of slow peers. Unavailable peer data are handled in a best-effort manner. Completeness estimation is used to estimate the percentage of unavailable and missing data and thus, helps to rate the significance of partial results. The idea of regularly maintained data summaries is a very interesting possibility to extend our approach. In Section 8.2.5 we evaluate the correlation between data and peer level on an empirical basis.

Concerning P2P query processing, the most related work is PIER [100]. The authors propose a similar database-like style of query processing as we do (see Section 6.6). However, until now, PIER provides no mechanisms for estimating query completeness, but the authors mention this for future work. ObjectGlobe [36] monitors query plans, so completeness estimation should be possible. But, the authors do not focus on that. Further, ObjectGlobe relies on centralized instances. In general, most approaches mentioned in Section 6.6 lack discussion about guarantees and predictability, such as estimating query completeness. [123] discusses (probabilistic) guarantees for P2P systems in general. But, the authors do not explicitly consider guarantees for query processing. Nevertheless, it is an interesting work concerning guarantees and predictability in P2P systems in general. Similar holds for [38], which states that in large-scale distributed systems *everything* can only be probabilistic.

Part IV

Evaluation and Conclusion

Chapter 8

Implementation and Evaluation

The concepts, algorithms, and techniques discussed throughout this work have been implemented in a prototype called *UniStore* [108]. That prototype was and is used to evaluate practicability and benefit of special methods and approaches. Section 8.1 gives a brief overview of the implementation and highlights several special issues. UniStore is primarily intended to provide a reference implementation that supports the evaluation of the proposed concepts. An extensive evaluation of the main concepts on the basis of UniStore is matter of Section 8.2.

8.1 UniStore: A Light-Weight Internet-Scale Universal Storage

We omit a detailed description of the implementation and the class hierarchy. The concepts presented in this work are all implemented straightforward. A general overview of the software is presented in Section 8.1.1. Throughout the work, we mentioned some special issues that result in certain special requirements on the implementation. We briefly discuss such issues in Section 8.1.2.

8.1.1 Overview of Implementation

UniStore is completely implemented in Java. This first increases its portability and provides a very light-weight and thin client. Further, this allowed us to tightly couple it with P-Grid, which is implemented in Java as well. This was especially useful as P-Grid is under steady development. Like this, we were able to use the latest features, identify and report bugs, and in places extend its code by some own lines. Nevertheless, UniStore is designed to work with any available DHT implementation.

The architecture of the implementation conforms to the layered architecture proposed in Section 3.3. The single components are implemented in separated packages. This encapsulation enables an extensible implementation, in which single parts can be modified and extended independently. The P-Grid code is encapsulated in one such package as well. Figure 8.1 provides a general overview of the main packages contained in the implementation. The names of the packages stem from the history of the implementation. We started with discussing similarity queries for P-Grid. Thus, the first packages reflect this in their names. We soon realized the challenging issues and promising features of the approach, which resulted in expanding the research activities to the focus of this work. Consequently, the first prototype was expanded to form the current UniStore software package. A current version of UniStore including source

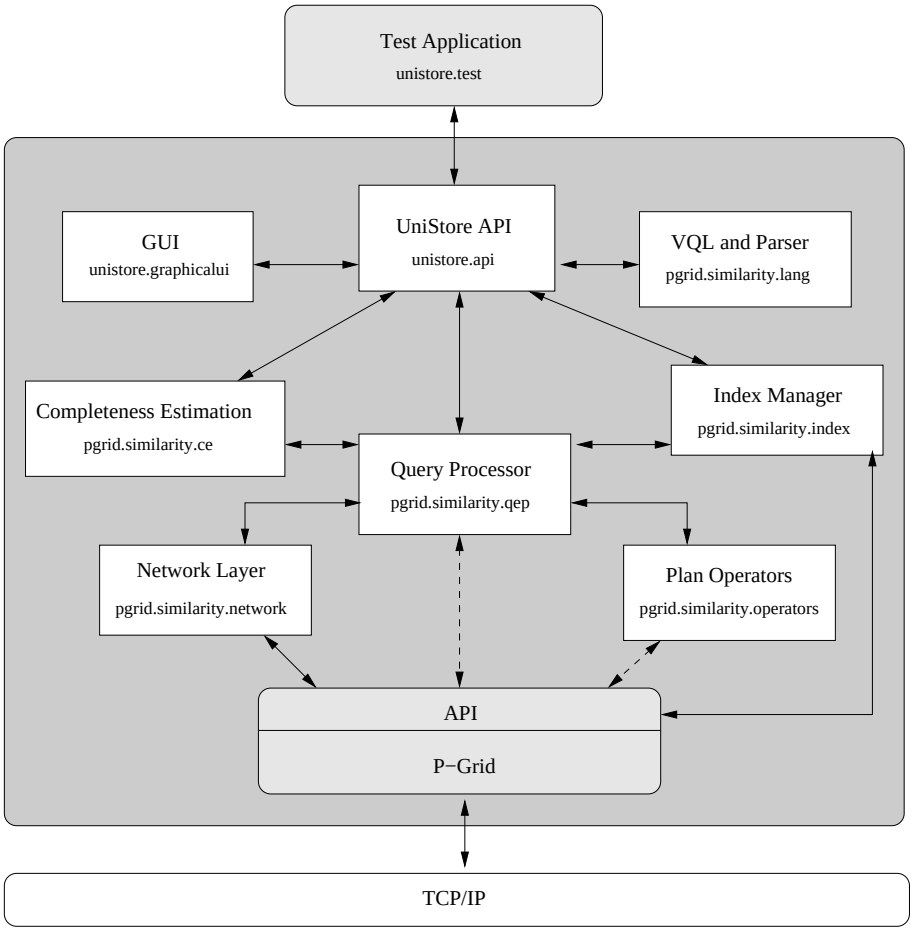


Figure 8.1: Implementation overview

files (licensed under extended GNU GPL) and further documentation can be obtained from http://www.tu-ilmenau.de/fakia/The-UniStore-Project.6485.0.html[1].

The center of the software package is composed of the query processor, which is the heart for managing and processing query plans, and the UniStore API, which represents the interface to the application layer. The arrows indicate which components use and rely on each other. As indicated in this work, different layers and components should be implemented in separated packages. The figure indicates this for the indexing facility, the completeness estimation, VQL level, etc. A physical operator conforms to executable code that is shipped to remote peers. These operators are implemented as state machines and implement the well-known iterator concept. Multiple plan operators form a query plan. As each instance of an operator class may store data internally and can have 0 to 2 children, no further construct is needed to represent query plans. In conjunction with a so-called plan wrapper, in which additional information like statistics can be shipped, these classes implement the concept of M^2QP. Variables used in query plans are implemented by exactly adhering to the concept of VTPs. They are represented in a special class. Instances of this class can store additional information, which is used to implement the $ref()$ and $oid()$ functionalities needed for query processing. The dissociation of the completeness estimation package from the plan operators allows for flexibly extending the

[1] last visited 2009/02/17

set of operators. As long as operators adhere to the interfaces defined by the query processor, this needs no further modifications. If new routing methods are included, this requires according modifications in all of these packages.

The interface to the P-Grid DHT is mainly encapsulated in the network layer package. All communication with the TCP/IP layer is delegated to the P-Grid system. Routing processes, incoming results, etc. are actually all handled in this component. Nevertheless, the query processor and the plan operators make in cases direct use of the P-Grid API (indicated by dashed lines). This is due to the mentioned steady development of P-Grid and special functionality that required small modifications. If these modifications and features are integrated in the steady development of both projects (as former issues were used to), the P-Grid package can be totally encapsulated from the UniStore package for official releases.

The presented architecture misses any components of the proposed semantic layer. This layer is not fully implemented yet. Since we focus on query processing issues and aim for providing the functionality for data integration tasks, the implementation though is suited for a meaningful implementation. The proposed map operator is included in the language level (VQL and parser) as well as in the level of physical operators (Plan operators). A basic mapping facility for handling schema correspondences and the actual mapping process is currently included in the physical operator.

The depicted test application was used for running the extensive evaluation presented in Section 8.2. We just had to distribute the UniStore package and the test application to machines that support Java and were enabled to run arbitrary tests. Configuration and other input files were used to control the actual test runs. The statistics were extracted from the plan wrappers that contain the actual query plans. Corresponding classes for managing these statistics are implemented in the query processor package. As a consequence, statistics can only be analyzed for query plans that actually return to the query initiator. Plans that get lost are simply excluded from the determination of statistics. Nevertheless, such lost query plan messages can be detected for evaluating the approach for completeness estimation, availability, and similar concepts.

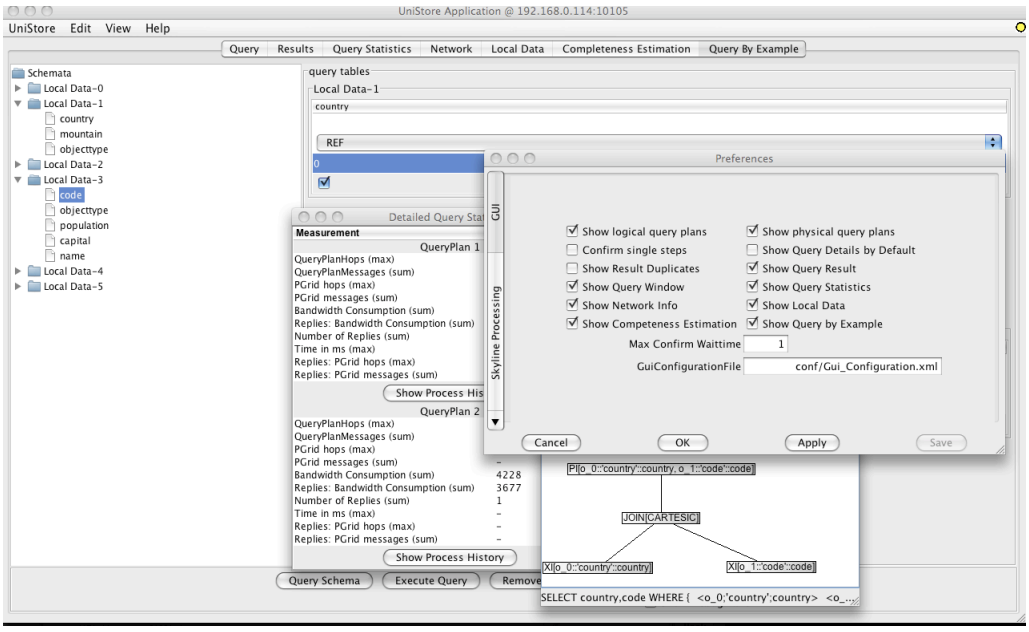


Figure 8.2: GUI screenshot

The UniStore software comes with a user-friendly GUI. This graphical interface allows for inserting data, formulating queries in VQL or supported by different "wizards" like a query-by-example engine, execute queries, view results, etc. It was designed to provide a light-weight and thin client for enabling Public Data Management applications. Thus, it supports special features like querying schema data. Further, it allows for evaluation and analyzing purposes, because it supports the illustration of query plans, query plan statistics, modification of variables and performance factors, etc. Figure 8.2 shows a screenshot of the GUI.

The UniStore software emerged with the support from several students. [210] implemented first basic operators and the hashing concepts that are still applied. Some main functionality of the query engine was implemented by [174]. This is the main reference for a detailed description of the main ingredients of the implementation. The implementation of skyline operators is described in [142], a basic functionality for supporting semantic correspondences was implemented by [138]. All these works also present some evaluation and a basic discussion of concepts. The implemented completeness estimation was developed with the valuable help of Michael Haß and Thomas Kreyling.

8.1.2 Special Implementation Issues

In this section, we briefly summarize some special issues of the implementation. First of all, there is the question of enabling the needed order-preserving hashing. Throughout this work, we always assumed an according implementation as provided. A closer look at this issue reveals some challenges. Popular order-preserving hash functions for strings are mostly based on prefix trees. This basically fine and also used in UniStore. But, this requires prefix trees that represent the managed data "preferably good". Discussing this and other approaches for general data sets is an interesting issue. Note that this is no problem for the evaluation in Section 8.2, because used test data sets can be analyzed locally before running tests. A balanced trie is constructed from a sample database that contains unique sample strings. This method was adopted from the P-Grid implementation. More details can be found in [188]. For enabling really flexible Public Data Management, this issue has to be solved in a more generic way. Similar holds for order-preserving hashing of numerical data. Currently, we use a rather simple approach: we use the binary string representation of a number for generating hash keys. For supporting order preservation, the used string representations have to be of equal length. This works fine, but should also be replaced by a more generic approach for supporting universal applications and data.

Representation used in the triple model are implemented straightforward. As mentioned before, a special Java class is used to implement the variable notation of VQL queries and query plans. Instances of this class are also used to provide the $ref()$ and $oid()$ functionality. A triple set consists of the raw triple data and according VQL variable references. For each variable, references to the according parts of triples are maintained in the triple sets.

Plan wrappers are used to encapsulate query plans, data, and additional information. Maybe the most important is the information used for estimating query completeness. Further data is not mandatory to enable concepts proposed in this work, but used for other issues. For instance, so-called plan rewriters can be included into plan wrappers to enable dynamic query planning. Further, instances of a query statistics class are included in plan wrappers. At each peer that handles a query plan message, these statistics are maintained accordingly. This enables a detailed evaluation of the applied concepts and operators. This required small modifications in the P-Grid code, because messages and hops needed for intermediate routing processes are logged as well. Note that the size of such additional information is not included into the determination of bandwidth consumption.

8.2 Evaluation

The purpose of the following evaluation is to provide a proof of concept for the discussed query processing techniques and approaches. We give more insights into the performance and characteristics of the different processing strategies and operators. A large set of experiments was run, here we present a selection of the most significant ones. The used setups and environments are described in Section 8.2.1. The realistic data used in the experiments is introduced as well and we briefly evaluate the implied storage overhead. Section 8.2.2 summarizes the metrics used in the evaluation and explains the general principles of the presented analysis.

The main objective is to prove the performance considerations discussed throughout this work. This goes along with evaluating the robustness and feasibility of the different processing concepts. This is matter of Section 8.2.3. We do not compare the distributed processing approach against a data shipping solution. A detailed analysis of the advantages and disadvantages of both approaches is matter of several other works. We expect a significant difference in the bandwidth consumption, whereas the number of generated messages and hops should be conform with the query shipping technique. Further, we do not provide an explicit comparison to a local processing approach. Nevertheless, the benefits of distributing query load gets evident throughout the evaluation. Concerning the strategies of this work, we expect parallel approaches to be more expensive than sequential ones, but in turn result in an increased robustness and shorter query answer times. The hash-based approaches should depend on the size and character of input data, but be rather independent from the size of the network. Special implementations like q-gram operators for similarity queries are expected to be superior in most situation, whereby the actual performance again depends on the type of data and query.

We also give insights into the appropriateness of the proposed cost model. But, this can be achieved only in limits. The main problem of estimating complex query plans are already known from traditional and distributed databases. The challenging task of estimating accurate selectivities and sizes of intermediate results can quickly result in estimation errors. the more complex a plan, the more intensive are errors propagated [101]. Thus, we exemplary focus on single operators and rather simple query plans. We expect the estimated costs reflect the actual costs very well. The most important point is that differences between operator alternatives can be evaluated meaningfully. Based on the cost model, we can also indicate the expected performance of processing strategies with much larger networks as physically available. The theoretical look on the performance and benefits should conform to the results gained in the experimental setups. Note that we do not run any simulation. All tests are executed in physically distributed network systems. Only the cost formulas are used to enable a rather theoretical evaluation.

In Section 8.2.5, we focus on the applicability, exactness and quality of the proposed techniques for estimating query completeness. This involves an evaluation of the proposed method for estimating range queries in P-Grid and the consideration of complex general query plans. Finally, we take a brief look on the performance of integrating semantic correspondences into query processing in Section 8.2.6. A final conclusion and a summary of outcomes and cognitions are presented in Section 8.2.7.

8.2.1 Experimental Setup

In this section, we briefly summarize the main facts about the setups used for the experimental evaluation. This comprises the physical environment, the handled data sets, and the test queries that are used.

Environment

In the beginning, we ran simplified simulations to give first insights into the performance and applicability of certain query processing techniques [106]. The promising results motivated to implement the full range of concepts in the UniStore prototype described in Section 8.1. The light-weight character and portability of the software allowed to run experiments in real-world setups, rather than struggling with the challenges of enabling a meaningful simulation. With Planet-Lab [50] as a test-bed, issues like dynamics and failures come for free. Planet-Lab is a world-wide consortium created especially for running large-scale distributed experiments. It currently consists of more than 700 nodes and is specifically dynamic and unreliable. Each participant can run widely distributed experiments on a subset of all Planet-Lab nodes. But, Planet-Lab is developed for evaluating network issues rather than for running large-scale data-oriented experiments. As a consequence, resources (particularly main memory) available at the participating nodes are very limited. Therefore, Planet-Lab bears several challenges for running meaningful large-scale experiments on a database background. Several of these issues have been observed by the designers of PIER as well [98].

For systematically evaluating operators and processing strategies while considering all aspects of real-world environments, we chose rather small subsets of all Planet-Lab nodes. We tried to gather specifically reliable nodes (steadily online and available). But, in each test run a fraction of nodes still failed and message delays and failure rates varied. Thus, we were enabled to base these experiments on a widely distributed setup, which bears all challenges of failures and dynamics, but still provides meaningful results. The number of involved nodes ranges from 30 to 120. Completeness estimation should also work for very unreliable networks. Since performance and availability are only secondary for evaluating the proposed approach, we also ran according tests in full-scale Planet-Lab setups. For this we gathered as many nodes as possible, resulting in setups with up to 460 nodes.

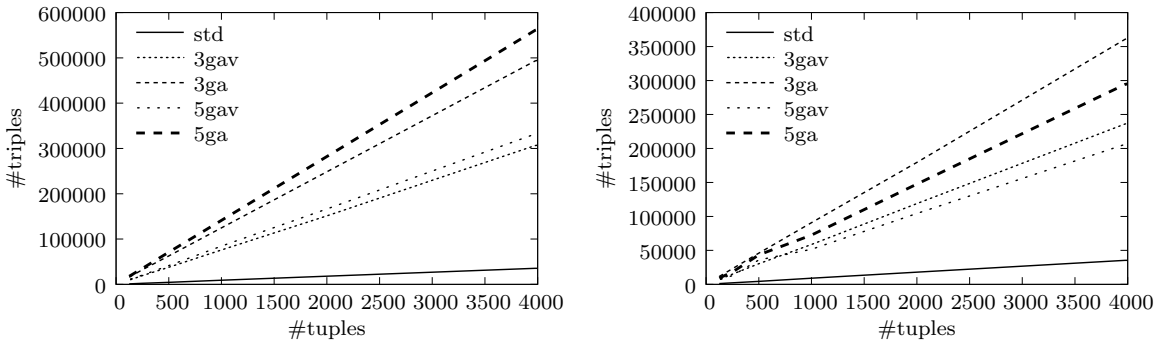
As fine as the Planet-Lab test-bed is for evaluating real-world applications in a realistic setup, the lowered is its suitability for discussing performance and quality of query processing in general. To gather more insights into issues and performance of the presented approaches, we also set up a local and reliable, but still realistic environment. We were able to run up to 74 nodes on a set of local desktop machines with meaningful performance. These machines were all connected by the local network and each ran up to 4 UniStore instances listening on different ports. As the environment was stable, we were able to use a low replication factor. This means, the number of peers replicating the same P-Grid path was lower than it would be used in more dynamic networks. Consequently, this resulted in a P-Grid tree as wide and as deep as it would occur in larger networks. The performance and quality of query processing depends on the number of participating peers only indirectly. Directly, it corresponds to the size of the underlying P-Grid tree. This, in turn, depends on network size and replication factor. The shape of th tree depends on the inserted data. We set the P-Grid replication factor to values between 1 and 3. In unreliable systems, this factor will be set to 5 or higher for compensating frequent joins and leaves of peers. In the Planet-Lab experiments we even used values larger than 5.

Data

Throughout the different test cases, we used different data from real world applications. To get a realistic representative of Web string data, we gathered data from the Internet Movie Database (Imdb[2]). The data is a collection of movie titles with some additional information

[2] http://www.imdb.com/

like IDs and year of production. It shows a skewed heavy-tail key distribution (power-law like). We modeled this data in different generic schemata. For this, we generated according attribute names. In different schemata, several different attributes refer to the same real-world concepts, which includes schema heterogeneities. Heterogeneities on instance level come for free due to the character of the used data.



(a) With special characters

(b) Without special characters

Figure 8.3: Q-gram storage for DBPedia and Mondial data

A data set representing typical data from Public Data Management scenarios is composed of triples from DBPedia[3] and geographical data in relational format from Mondial[4]. These data are taken from a realistic scenario combining geographical data from both sets.

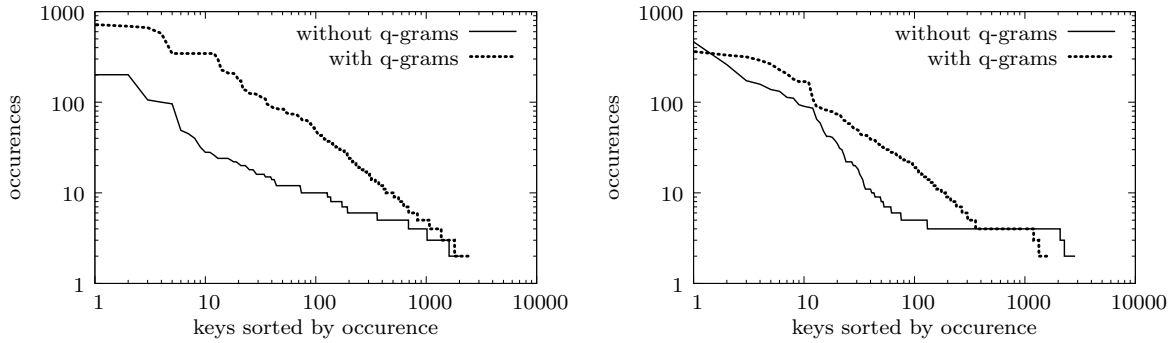
The resulting average and maximal lengths of P-Grid paths was between 3 and 5, between 5 and 8 respectively. This holds for the local setups and the Planet-Lab setups with up to 120 peers. As these lengths increase with the number of participating peers (and data), we observed accordingly larger values for full-scale Planet-Lab experiments with up to 460 nodes. To be able to deduce to large networks from rather small setups, we tried to let these values approach each other by lowering the P-Grid replication factor for smaller networks.

To evaluate the storage overhead of the vertical approach, we plot the number of generated index items on the basis of the three standard indexes from Section 4.2.1 in Figure 8.3. In the figure, the according amount of triples is represented by `std`. The number of triples resulting from each input tuple is independent from the contained data. Thus, the number of index items for these three indexes is independent as well. If only one index would be built, the number of generated index items would be exactly three times smaller. Additionally, we illustrate the overhead generated by q-gram indexes, which depends on the actual data and the size of q-grams (the q). We built both introduced q-gram indexes (on attribute-value basis and on attribute basis) on all inserted triples. `3ga` stands for the number of index items resulting from attribute and attribute-value indexes, `3gav` for only the attribute-value index. Also, the results for indexes with q=5 are depicted. Further, we differentiate between the inclusion and exclusion of special leading and trailing characters used for extracting q-grams. Results for the Imdb data look very similar, albeit the overhead is smaller, because attribute names and values are shorter.

Building q-gram indexes results in a noticeable storage overhead. The exact overhead depends on the chosen q and the decision on the usage of special characters. With special

[3] http://dbepdia.org
[4] http://www.dbis.informatik.uni-goettingen.de/Mondial/

characters, the overhead increases with increasing q. Without special characters, this is vice versa. In the experiments presented in this evaluation, we used a value of 3 for q and included special characters. The decision whether to accept these additional storage costs depends on the benefit of the q-gram indexes. After discussing the experimental results in the next sections, a rational decision will be enabled.



(a) DBPedia and Mondial data

(b) Imdb data

Figure 8.4: Key distribution

Figure 8.4 additionally shows the resulting skewed distribution of triples from two example runs. The figures look analog for all other test runs. We plot the number of occurrences for all keys that were inserted more than once on a double-logarithmic scale. The q-gram attribute and attribute-value indexes were built with the inclusion of special leading and trailing characters, q was set to 3. Here we show also the changes in the key distribution for the Imdb data, which illustrates the smaller overhead the q-gram indexes cause on this data.

Queries

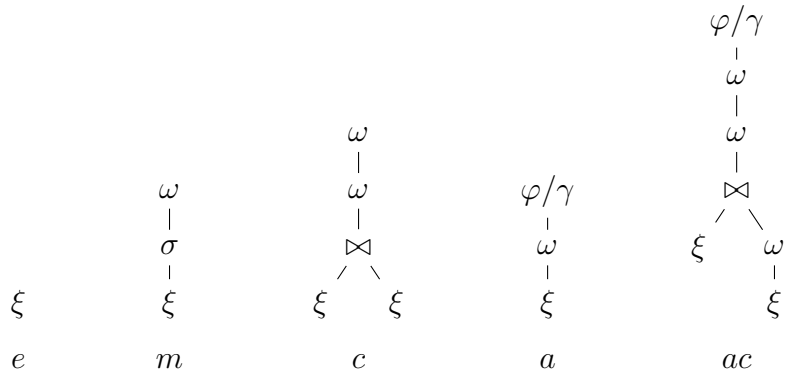


Figure 8.5: Shape of evaluation queries and class identifiers

The queries used in the evaluation can be assigned to general classes as shown in Figure 8.5. The figure illustrates the general shapes of five classes. We refer to these classes using the identifiers e, m, c, a and ac. In the figures and explanations we use corresponding symbols and abbreviations, which are introduced where needed. These query classes are used on all involved

data sets. In the following, we list an example query for each class. The actual queries vary in the used values for attribute names and attribute values, optional filter statements, the applied level(s) of querying (schema, instance, and OID level), and the exact number of operators (i.e., triple patterns). We highlight the specific differences accordingly.

- class e:

```
select ?n where {?x 'title' ?n filter edist(?n,'stranger')<6 }
```

- class m:

```
select ?v, ?c
where {?x 'title' ?v . ?x 'count' ?c
       filter ?c >= 30 filter edist(?v,'slice_the')<4}
```

- class c:

```
select ?xt, ?xl, ?yn
where {?x 'title' ?xt . ?y 'n' ?yn . ?x 'len' ?xl
       filter edist(?xt,?yn)<3}
```

- class a:

```
select ?v, ?r
where {?x 'date' ?v . ?x 'r' ?r } order by ?v desc limit 20
```

- class ac:

```
select ?d, ?l
where {?x 'title' ?k . ?y 'n' ?k . ?y 'r' ?r .
       ?x 'date' ?d . ?x 'len' ?l}
order by ?k NN 'Haven_The' limit dist 6
```

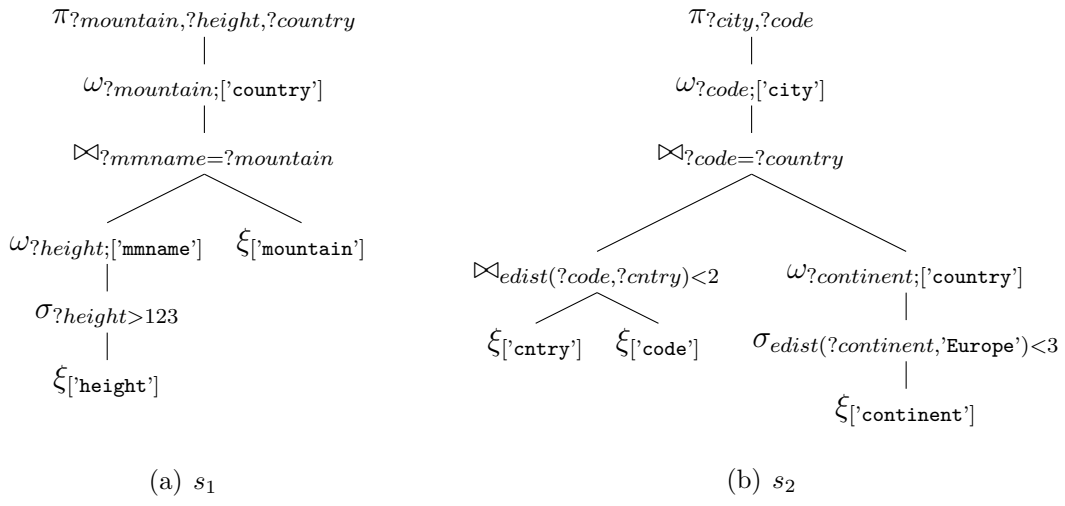
In addition, on the DBPedia and Mondial data we used two complex queries of class s that use mapping data. The original queries have a shape as the examples shown in Figure 8.6. These queries are expanded on the basis of simple mapping data (`map:equiv`) and on the basis of ontology data. Triples and queries for representing and querying semantic correspondences look as introduced in Sections 4.2.3 and 5.1.2.

Example queries conforming to the two classes are listed next. Here, actual queries differ in the expanded attribute(s) and whether ontology data is queried or only simple mappings.

- query s_1:

```
expand height
expand mmname

select ?mountain, ?height, ?country
where {?o1 'height' ?o1 . o1 'mmname' ?mountain .
       ?o2 'mountain' ?mountain . ?o2 'country' ?country
       filter ?height>123 }
```



(a) s_1 (b) s_2

Figure 8.6: Shape of queries for semantic expansion

- query s_2:

```
expand *

select ?city, ?code
where {?x 'cntry' ?cntry . ?y 'code' ?code . ?y 'city' ?city .
       ?z 'continent' ?cont . ?z 'country' ?code
       filter edist(?code,?cntry)<2
       filter edist(?cont,'Europe')<3 }
```

In the first series of experiments, we systematically compare the different operator implementations and processing strategies. For this purpose, we generated a query mix of 119 test queries. These queries span from simple extractions to complex join and rank queries. The actual shape of a query plan and the contained operators are more relevant than the actual query. Thus, at the according places we indicate the class that the used query plans belong to and mention specific variations.

We designed a query mix for the DBPedia and Mondial data that is particularly suited to evaluate the quality of estimating query completeness. The queries in the mix were chosen in order to be representative for both routing methods introduced and to span from simple to rather complex queries. For processing the queries, we varied the used physical operators. The queries are chosen to conform to a real-world scenario. Beside attribute lookups, they include similarity joins and selections based on q-gram operators, as well as aggregation operators and ranking operators.

All queries with all variations were run multiple times and the average values are presented.

8.2.2 Metrics

The implemented prototype allows for collecting a wide range of statistics. The most important performance factors are the number of messages, the number of hops, and the total bandwidth consumption a query needs. Thus, we focus on these three measures. Concerning messages and

hops, we can differentiate between logical M^2QP level, physical M^2QP level, and P-Grid level. The difference is:

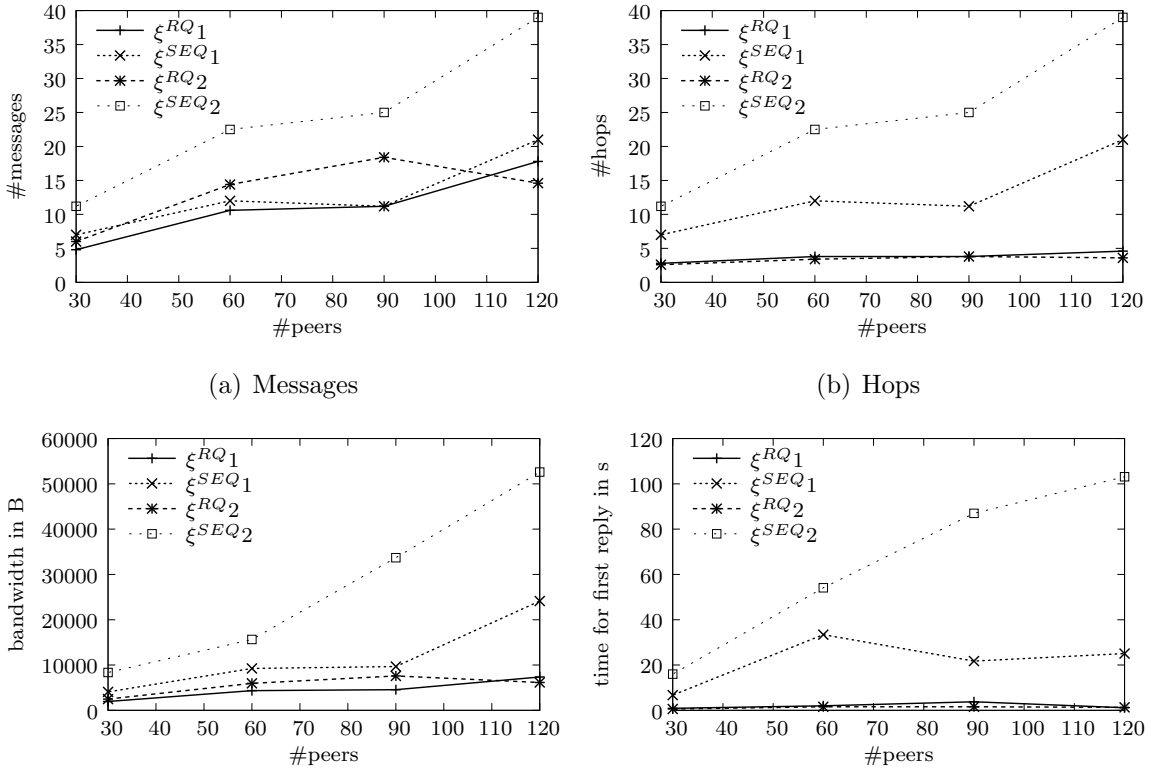
- each time a routing process for a query plan (copy) is initiated, the number of messages on logical M^2QP level is incremented
- if this results in a plan copy that is physically sent over the network (a peer may "route" a plan to itself), the number of physical M^2QP messages increases as well
- for each routed query plan, there will be several intermediate forwarding peers on P-Grid level – thus, the numbers on P-Grid level are always above the other two levels

Consequently, the logical M^2QP level is best suited to show the accuracy of cost estimations. The physical M^2QP level is suited to illustrate the effort for the different processing approaches independently from the underlying DHT. In all tests we observed that trends in both M^2QP levels are always reflected by according trends in the P-Grid level. Since the idea of using a DHT is a main ingredient of the proposed framework, the P-Grid level is the most significant for performance evaluation. Consequently, in the evaluation we focus on the P-Grid level. Bandwidth consumption is determined only on P-Grid level and usually corresponds to the number of messages. We get back to this measurement only when there are meaningful differences.

The robustness of different approaches in the presence of failures and delays can be expressed by the fraction of expected results that is actually received. To evaluate this, we compared the received results to the results we would get from all inserted data (determined locally). The resulting fraction is called is called availability throughout this evaluation. For evaluating completeness estimation, we compare the estimated completeness on peer level with the actual completeness on peer level (fraction of already received replies). To judge on the feasibility of the peer level approach, we additionally use the actual completeness on data level for comparison (fraction of already received results).

Average and maximal response times usually correspond to the number of hops. A more significant measurement is the time for first replies, because this evaluates the suitability for the preferred online processing approach. Answer times are discussed only when they help to identify meaningful cognitions. In some cases we show the development of the number of replies or results (contained in replies) over time. This helps to illustrate answer times as well as overhead. The higher the degree of parallelism, the more replies will be received and the more messages are needed. The number of replies also helps to deduce on the bandwidth consumption that is needed for only reply messages. Sequential approaches should return the same amount of results as parallel approaches, but in significantly less replies, which arrive later than replies from queries processed in parallel.

In most other cases we focus on the scalability of query processing by presenting results with respect to the number of participating peers and handled data volume. We either increase both in parallel or keep the data volume constant while increasing the number of peers. In Section 8.2.6 we briefly show that the differences between both variants are insignificant. In places we also consider increasing query load and complexity for queries of the same class, e.g., by increasing the number of materialized attributes and queried distances in similarity queries. Different data distributions are represented by the different test data, failure rates and message delays come for free due to the usage of the realistic environments. Different complexities of VTPs and different levels of querying (instance, schema, OID) are covered by the used query mixes.



(a) Messages

(b) Hops

(c) Bandwidth consumption

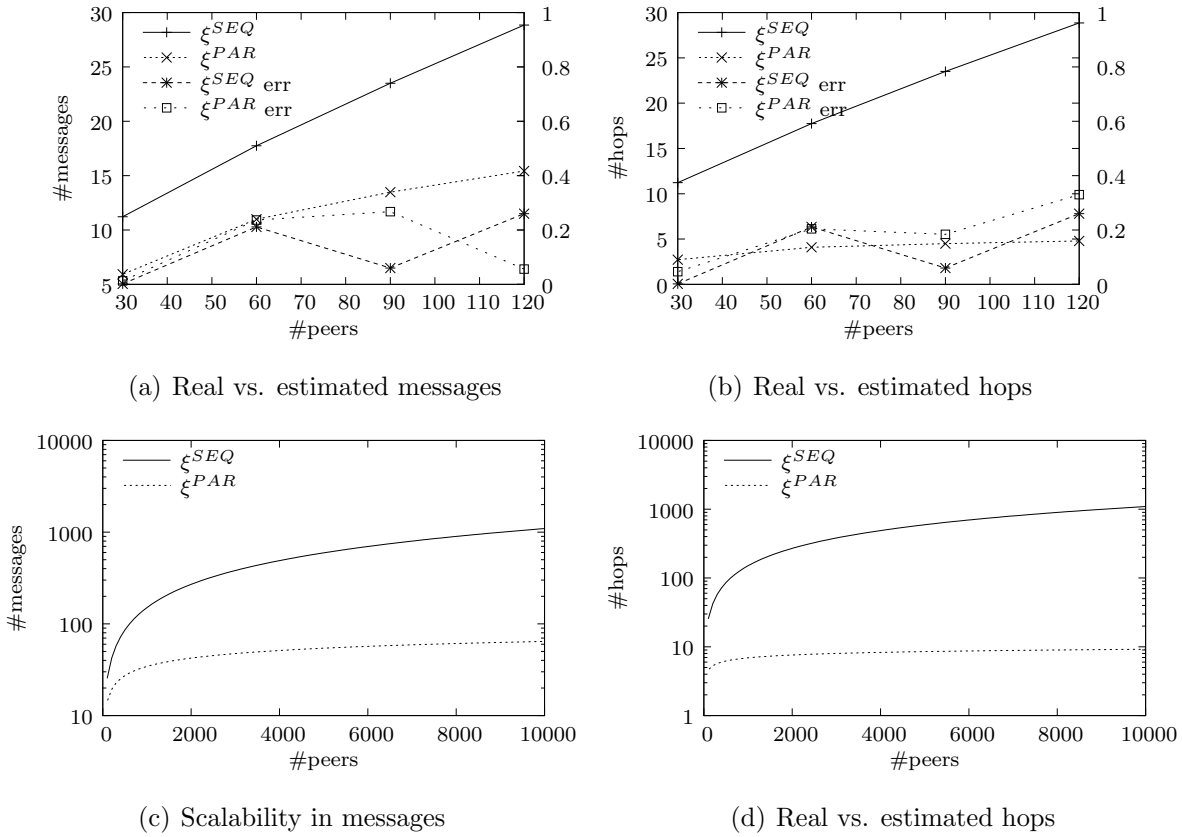
(d) First replies

Figure 8.7: Comparison of ξ^{RQ} and ξ^{SEQ} for two different VTPs

8.2.3 Physical Operators and Processing Strategies

In the following part of the evaluation we focus on systematically comparing the different operator implementations and processing strategies. The tests were run on 30, 60, 90, and 120 nodes from the Planet-Lab test-bed. We used the Imdb data and inserted 5 tuples at each node. We built the full range of introduced indexes: an OID index, an attribute-value index, an attribute index, and two q-gram indexes. The first q-gram index was built on the values of one attribute from each original schema, the second q-gram index on schema level for four attribute names from each input schema. Each peer generated about 200 index items for the 5 inserted tuples. The number of items that belong to the OID index and the attribute-value index is 30 for each, the remaining items belong to the q-gram indexes. Like this, the size of the handled data rises with the number of participating nodes, ranging from about 6000 index items to about 24.000 index items. Unfortunately, we were not able to increase the size of inserted data while still getting significant results, due to the strict limitations on Planet-Lab. Nevertheless, the accomplished results are suited very well to prove the theses of this work. In places, the results gathered from the runs on 90 nodes show an abnormal increased overhead compared to other runs. But, this holds for all of the test queries, which still allows for a meaningful comparison. We were able to run most of the queries five times in one run. More repetitions were only possible by distributing them over several runs, due to the instability of Planet-Lab setups.

In general, we expect parallel operators to show a significantly improved robustness compared to sequential approaches. This should come along with an increased effort, represented



(a) Real vs. estimated messages

(b) Real vs. estimated hops

(c) Scalability in messages

(d) Real vs. estimated hops

Figure 8.8: Cost comparison of ξ^{RQ} and ξ^{SEQ}

by more messages and higher bandwidth consumption. The increased robustness should be backed by lower reply times and decreased hop counts, but in turn result in more replies. The high message load of parallel processing approaches should make sequential approaches better suited for specific situations and queries. These differences should conform to the cost model introduced for physical optimization.

This section is structured according to the different operator classes. We begin with basic routing operators, get over to join and similarity operators, and briefly conclude with special operators for ranking queries.

Basic Routing Operators

First, we focus on implementations for the basic routing operators ξ and ω. [60] already showed that the min-max traversal strategy for range queries (corresponding to sequential extraction ξ^{SEQ}) performs worse than the shower algorithm (corresponding to range-based extraction ξ^{RQ}). We can back these observations. In Figure 8.7 we compare the number of messages, number of hops, bandwidth consumption, and time for first replies for two different extractions. The queries belong to class e introduced before. While the number of messages shows only small differences, the number of hops and the used bandwidth consumption prove the advantages of the parallel approach. The trend in the bandwidth consumption follows the trend in the number of messages – for the second query, which involves a larger range than the first one, the needed

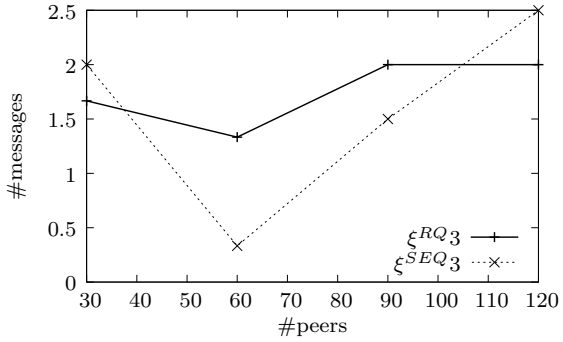
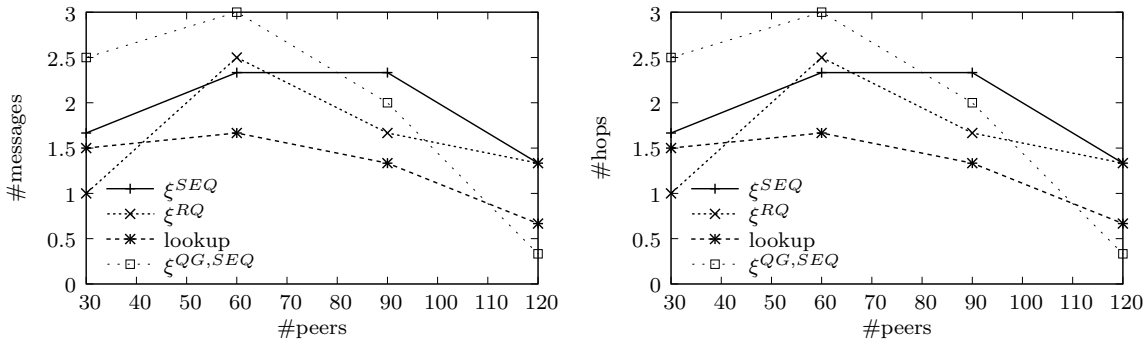


Figure 8.9: ξ^{RQ} vs. ξ^{SEQ} on small ranges

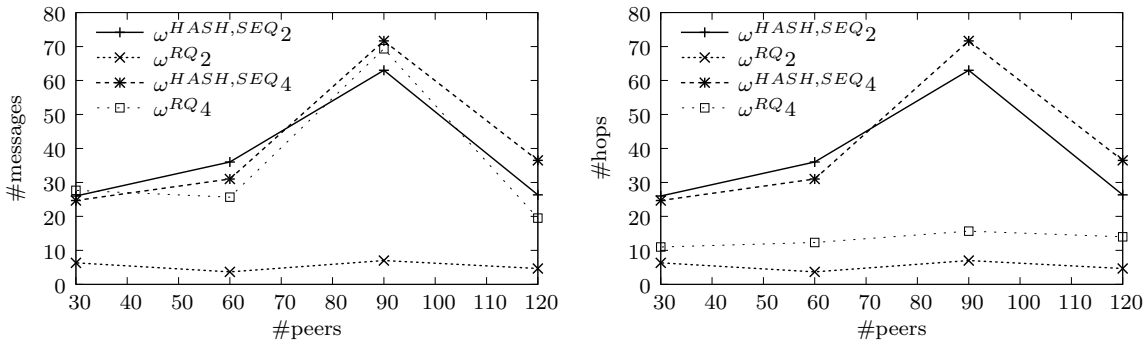


(a) Messages (b) Hops

Figure 8.10: Range- and hash-based extractions vs. lookups

bandwidth scales unacceptably. The fact that ξ^{RQ} should usually be chosen on leaf level of query plans if ranges are addressed is backed by the small reply times.

In Figure 8.8 we show the estimations that are gained from the proposed cost model. We plot the estimated absolute values (left y axis) and the resulting relative error of estimation (right y axis). Figures 8.8(a) and 8.8(b) show that the general differences and trends in the absolute values are reflected very well. But, a relative error from up to 25 percent is almost too high for a valuable cost estimation. As the errors do not seem to depend on the network size, we have to look for the reasons in other factors. A crucial factor is the failure rate of peers and messages, which decreases actual cost values. But, these issues are not covered by the cost model as introduced. Second, we already mentioned the difficult estimation of costs for processing range queries on the basis of P-Grid shower algorithm. The fluctuations in the estimation of according costs illustrates the importance of that issue. Beside issues of availability and failures, a main problem is the accurate estimation of the costs in *all* situations. This is already indicated by the $\lceil\lceil \ldots \rceil\rceil$ relations used in the cost formulas from Section 6.5. However, as the costs reflect the main differences between the operator alternatives, they are suited to fulfill the main task they were designed for, the decision on chosen operator implementations. This is why we also use them to simulate the scalability of the operators. The results in Figures 8.8(c) and 8.8(d) (note the logarithmic scale) confirm the basic trends that we identified in the experiments.



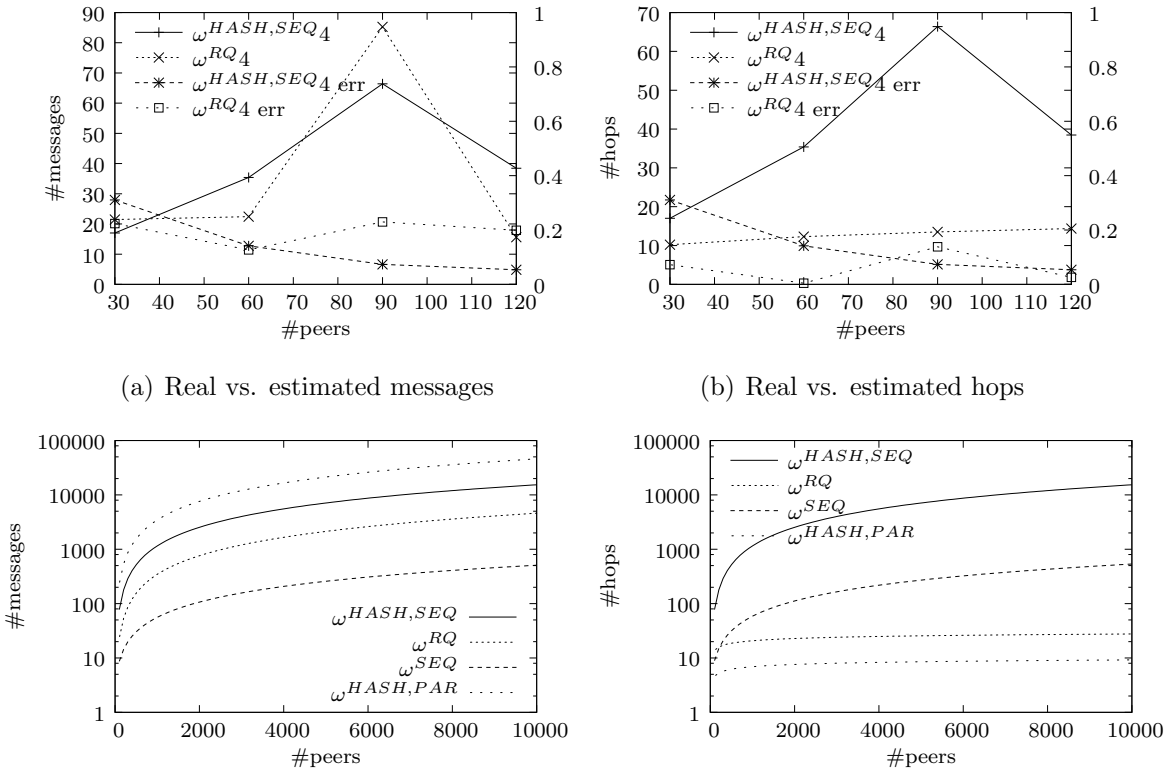
(a) Messages (b) Hops

Figure 8.11: ω^{RQ} vs. $\omega^{HASH,SEQ}$ for different numbers of materializations

The dependence of performance on the size of the queried range is illustrated in Figure 8.9. Here we show the possible overhead of range queries for very small ranges. This reflects what is expressed in the cost model from Section 6.5.
Intuitively, direct lookups should be applied if the provided VTP allows that. The overhead of other extraction operators, e.g., for similarity extractions, is shown in Figure 8.10. For the lookup we used a query from class e with an according VTP. For the other operators, we used e extended by a similarity selection σ. For the q-gram operator, which is the only hash-based operator, selection and extraction are combined to $\xi^{QG,SEQ}$. The resulting overhead compared to the lookup operation is rather small, the bandwidth consumption shows a similar behavior. The decreasing trend for 90 and 120 peers is surprising and unexpected. But, as all plots show the same trend, the comparison is not influenced. The overhead of the q-gram operator for smaller networks is due to the sequential character and the applied distance. We present a comparison to the parallel approach below. The figure also shows that hops and messages can show the same values in specific situations, particularly if addressed ranges are rather small. Average values can get smaller than 1 if queries can be (partly) answered locally.

As the q-gram operator indicates, sequential approaches can be suited for hash-based operators very well. In this case, they do not depend on a queried range. Rather, they depend on the set of input candidates, which can make them perform better than range-based operators. For materializations, this is particularly true if multiple attributes have to be materialized. Then, the hash-based materialization ω^{HASH} should perform better, because it utilizes the OID index. We show this in Figure 8.11. The queries are from class m, but materialize a different number of attributes: $m2$ materializes 2 additional attributes, $m4$ materializes 4 additional attributes. On leaf level, all query plans use ξ^{RQ}. The results show the issues on the 90 peer setups mentioned before. But, all queries show an according decrease in performance. The hash-based approach shows a clear overhead in the number of messages compared to ω^{RQ}. But, this overhead decreases with increasing number of materialized attributes. Surprisingly, even with four chained ω^{RQ} we result in not more messages than with ω^{HASH}. As expected, the number of hops of the parallel range-based approach is always below that of the sequential hash-based approach.

The ω operators compared in Figure 8.11 are the next that we use to illustrate the applicability of the proposed cost model. In Figure 8.15 we again indicate the general correctness of hash-based and range-based estimations. The figure shows the estimated absolute values and the resulting relative errors for estimating the costs of the query materializing 4 attributes. The results are slightly better than for parallel range-based operators, but still show some signifi-



(a) Real vs. estimated messages

(b) Real vs. estimated hops

(c) Scalability in messages

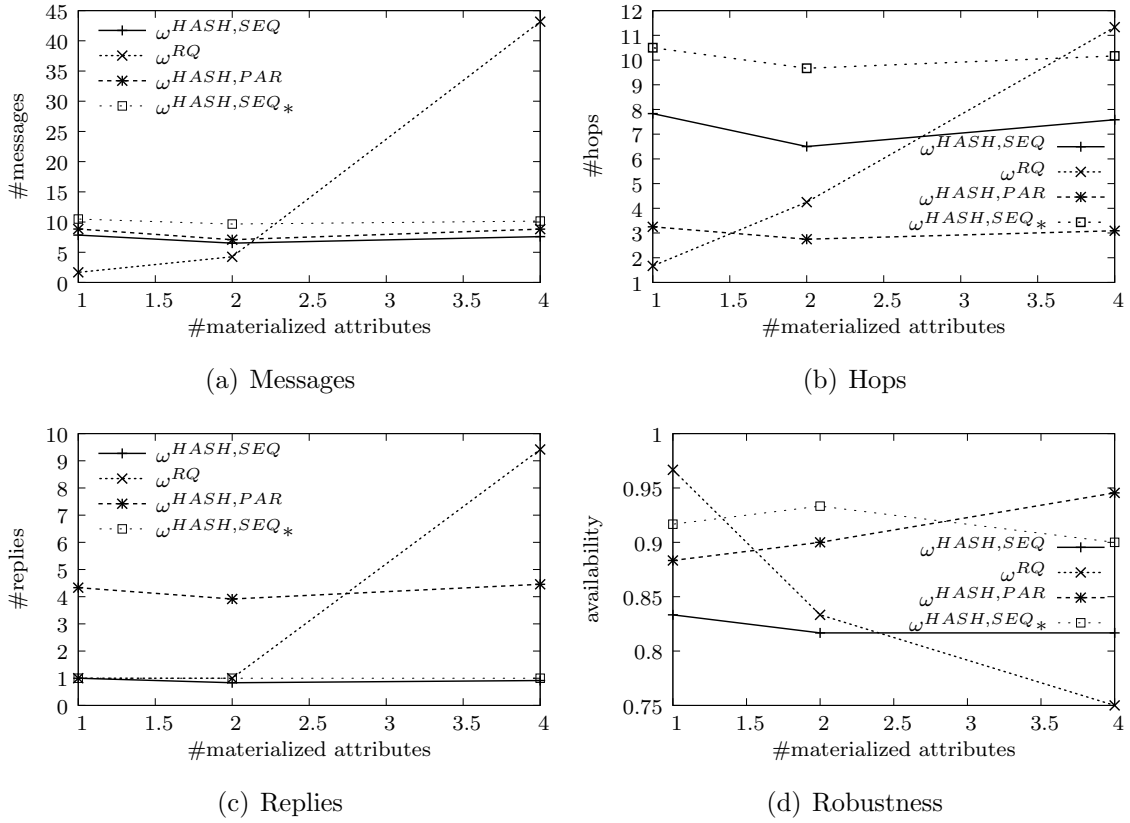
(d) Scalability in hops

Figure 8.12: Cost comparison of ω variants

cant peaks at about 20 percent error. Again, the observed trends are reflect in the estimated values. Thus, we again use the cost model to take a brief look at the theoretical scalability of the proposed approaches. We omit the estimation of hops, because the cost model estimates them identically for both used operators.

This advises to take a closer look on the influence of the number of materialized attributes. To illustrate the differences between the range-based and hash-based approaches more clearly, we chose queries from class m that address larger ranges than before. ω^{SEQ} is omitted, because we proved the disadvantages of sequential range queries before. But, this time we include the parallel hash-based materialization $\omega^{HASH,PAR}$ into comparison. Figure 8.13 contains the achieved results, with respect to the number of materialized attributes. The first observation is that the range-based approach performs significantly worse as the addressed ranges and the number of materialized attributes increase. This holds for the number of messages and the hop count. The fact that there is only 1 reply from ω^{RQ} for up to two materialized attributes implies that the ranges for both attributes are each covered by only one peer. This is different for the third and fourth attributes.

Number of messages, hop count, and the number of replies stay constant for the hash-based operators that utilize the OID index. As expected, the hop count for parallel operators is below that of sequential ones, which does not hold for the number of messages. We additionally show an important characteristic of sequential operators. As discussed in this work, sequential approaches can be used to reduce overhead by processing as much of an input set as possible



(a) Messages

(b) Hops

(c) Replies

(d) Robustness

Figure 8.13: Variants of ω for different numbers of materializations

at each peer. The impact of this is illustrated by $\omega^{HASH,SEQ}*$ in the figure, which corresponds to $\omega^{HASH,SEQ}$, but does not apply this optimization step.

Figure 8.13(d) highlights an important strength of parallel operators: the increased robustness. As messages can get lost, parallel operators result in a significantly better availability than sequential operators. The availability for range-based operators decreases with increasing number of materialized attributes, because all attributes have to be materialized in sequenced separate operators.

The time for first replies depicted in Figure 8.14 underlines the characteristics of the parallel approach. Unfortunately, we involved particularly slow peers in the single-attribute materialization from each run. But, this indicates another important fact: sequential operators are extremely vulnerable to single slow peers. Further, it shows that the number of materialized attributes is not relevant for the performance of hash-based materializations.

Figure 8.15 illustrates that the decision on the physical materialization operator usually dominates performance of a query combining ξ and ω. This can also be identified in Figure 8.27, which is discussed below. m^{SEQ} combines ξ^{SEQ} with $\omega^{HASH,SEQ}$, $m^{RQ,SEQ}$ combines ξ^{RQ} and $\omega^{HASH,SEQ}$, and m^{RQ} combines ξ^{RQ} with ω^{RQ}.

We could show that sequential range-based operators in most cases perform worse than those using a parallel range-querying approach. This backs what was already observed in other works. The hash-based materialization operators represent a valuable alternative to range-based operators. Whether parallel or sequential approaches should be chosen depends on the size of input data – and reflects in the number of generated messages and needed hops. In all cases,

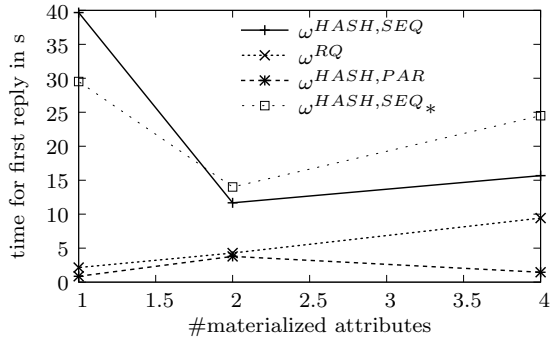
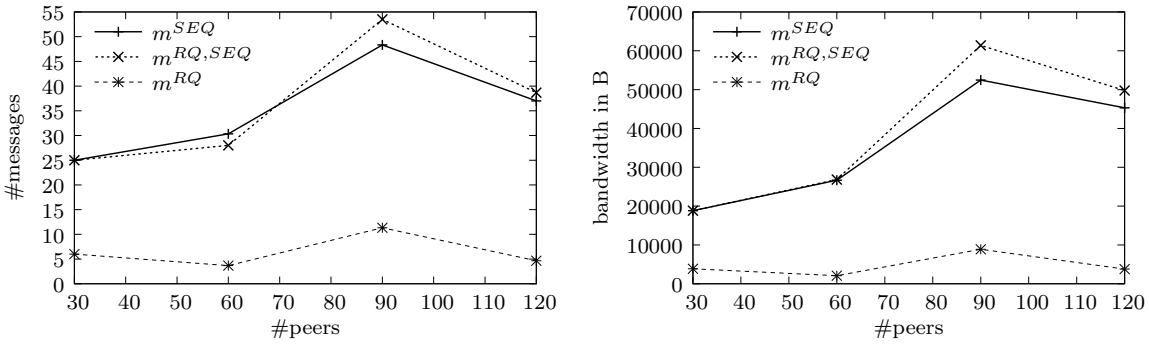


Figure 8.14: First reply times for different numbers of materializations



(a) Messages

(b) Bandwidth consumption

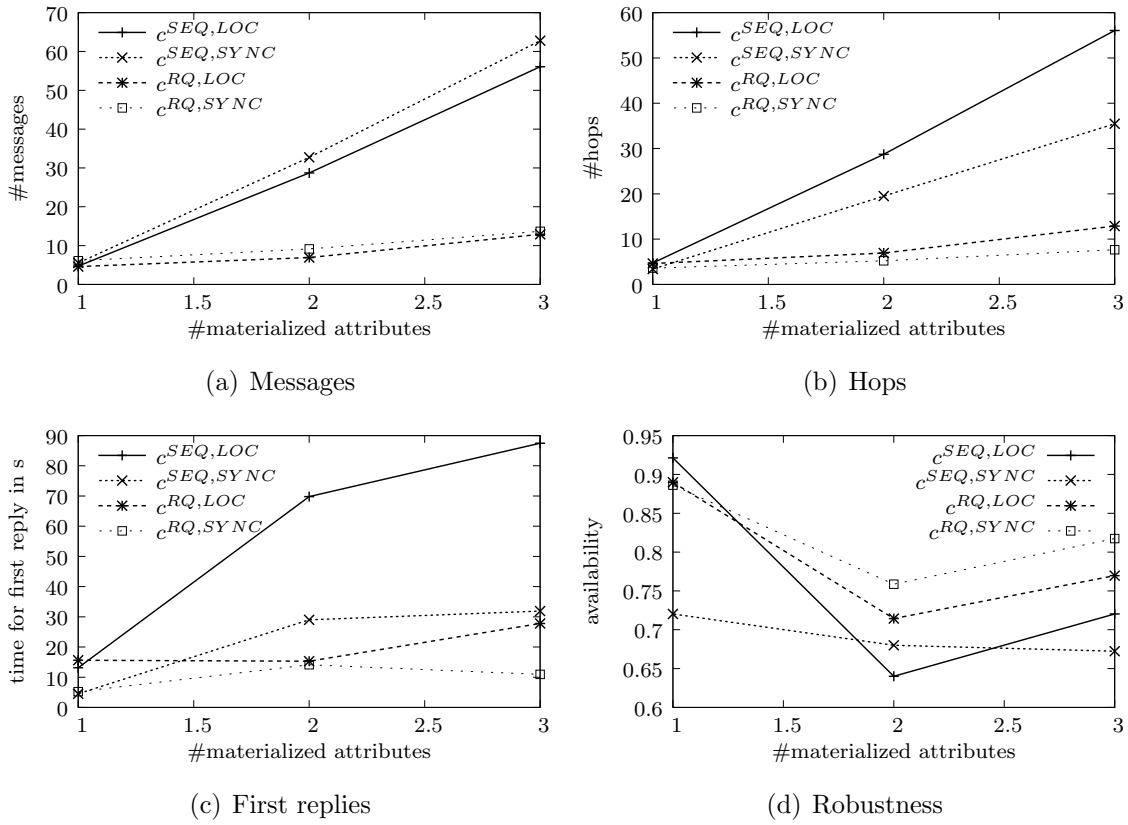
Figure 8.15: Combination of ξ and ω

parallel operators are particularly suited for unreliable and dynamic environments. Hash-based operators should be the preferred choice if the input data is small or if multiple triples have to be materialized for single input triple sets. The benefits of range-based operators decrease with an increasing number of participating peers. Operators like σ^{RQ} are not evaluated separately. They correspond to range-based extractions and materializations.

Standard Join Operators

Next, we get over to compare standard join implementations. As corresponding queries from class c always depend on the join implementation and the routing operators located below, we evaluate both in conjunction. In Figure 8.16 we compare local joins $\bowtie^{LOC}$ and inter-operator parallel joins $\bowtie^{SYNC}$. Each variant is combined once with ξ^{SEQ} and a varying number of $\omega^{HASH,SEQ}$ ($c^{SEQ,LOC}$ and $c^{SEQ,SYNC}$ in the figure), and once with ξ^{RQ} and a varying number of ω^{RQ} ($c^{RQ,LOC}$ and $c^{RQ,SYNC}$ in the figure). Results for the hash-based join $\bowtie^{HASH}$ can be deduced from the results for the other hash-based operators, such as the materialization ω^{HASH} discussed above and similarity joins $\bowtie^{HASH}$ (ξ^{QG}), which are discussed below.

The number of messages and hops are clearly influenced by the operators chosen for extraction and materialization. Although, the local join requires slightly more hops than the parallel variant, but less messages. This is as expected, due to the concept of synchronization. The parallel approaches result in more replies, which in turn result in a lower reply time and an



(a) Messages (b) Hops

(c) First replies (d) Robustness

Figure 8.16: $\bowtie^{LOC}$ vs. $\bowtie^{SYNC}$ for different numbers of materialized attributes

increased robustness, as Figures 8.16(c) and 8.16(d) show. The high reply time for $c^{SEQ,LOC}$ results from the fact that only one reply containing all join data is generated. As expected, the availability for all variants decreases with increasing complexity, i.e., with increasing number of materialized attributes. Surprisingly, the synchronizing join method performs worse than the local method for the sequential variant from this point of view. This is due to the extra messages needed for contacting the synchronization peer. In contrast, for parallel extractions and materializations, the synchronization step helps to increase robustness. This is because join candidates are gathered already after one range-query phase. For the local join, results can only be generated after two subsequent range-query phases. Of course, this requires reliable synchronization peers.

Summarizing, the observations for join operators are analog to those for basic operators. Parallel approaches, even with synchronization, usually result in an increased robustness and faster replies. This comes on the costs of more messages. The extraction and materialization operators used for generating the join input have a significant influence on the performance of the whole query.

Similarity Operators

Up to here, we discussed the main results concerning standard database functionality in the proposed query processing framework. Now, we extend this to include also the very important similarity operators. Performance of similarity queries is particularly dependent of the query

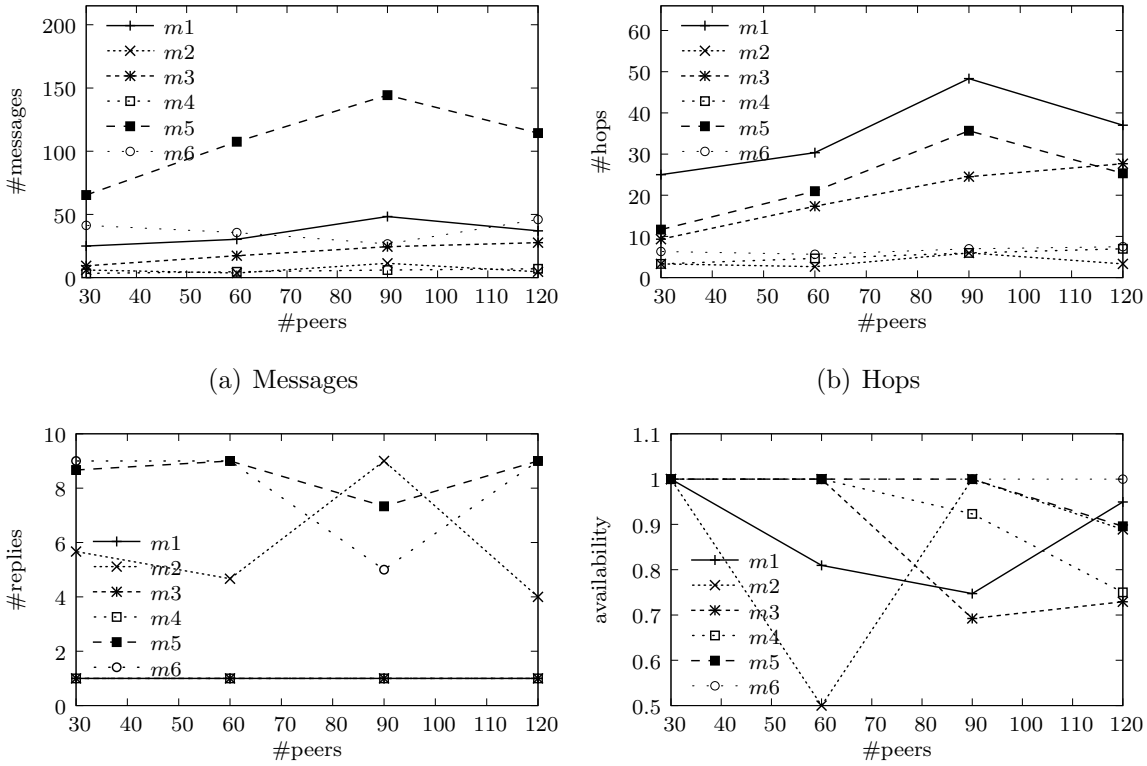


Figure 8.17: Similarity extraction

level (schema, instance, or OID level). Thus, we include an according differentiation in the following.

We first focus on the leaf level of query plans and analyze similarity extractions. Figure 8.17 compares sequential, range-based, and q-gram-based approaches suited for this. We ran a query of class m that includes one string similarity extraction and one standard materialization. We include the following variants into comparison:

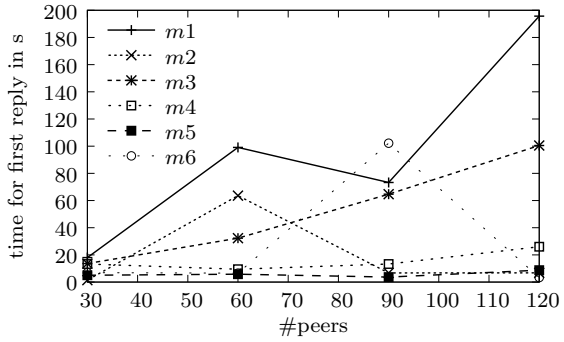
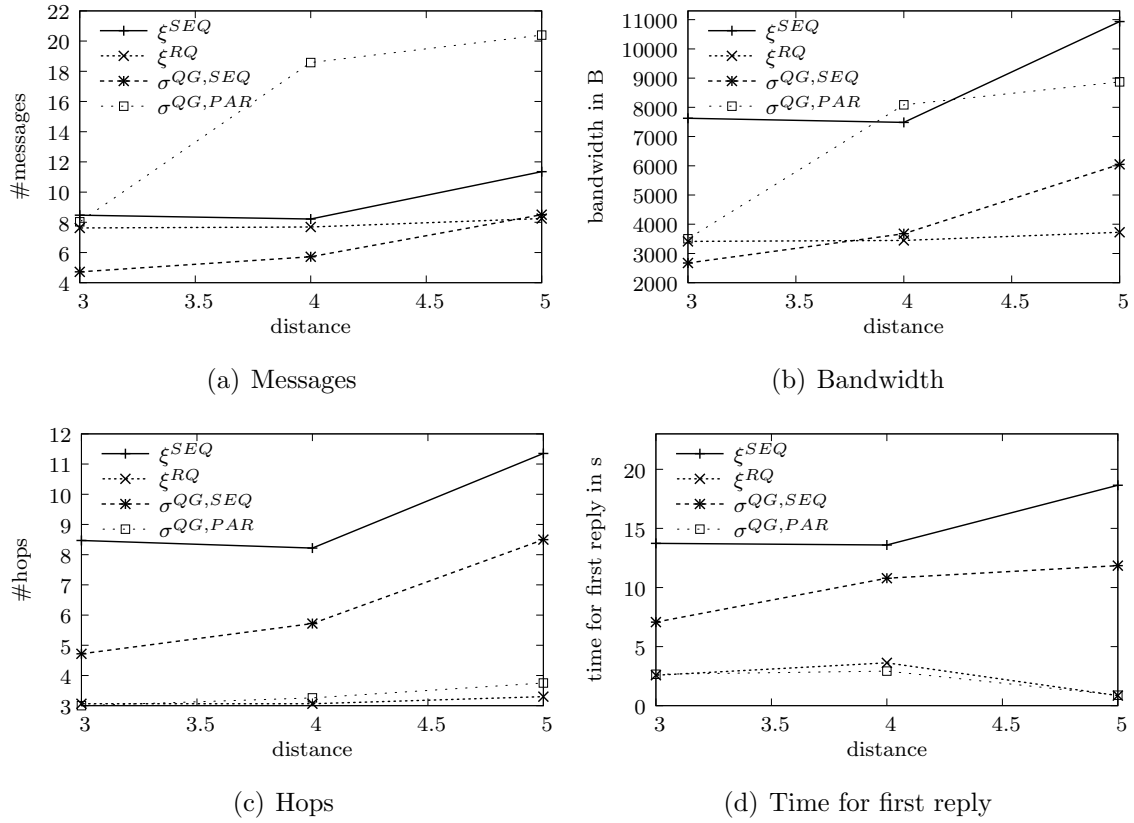


Figure 8.18: First reply time for similarity extraction

- $m1$: uses ξ^{SEQ}, $\omega^{HASH,SEQ}$, and a local similarity selection σ^{LOC}
- $m2$: uses range-based operators ξ^{RQ} and ω^{RQ} as well as a local similarity selection σ^{LOC}
- $m3$: uses $\sigma^{QG,SEQ}$ and $\omega^{HASH,SEQ}$
- $m4$: uses $\sigma^{QG,SEQ}$ and ω^{RQ}
- $m5$: uses $\sigma^{QG,PAR}$ and $\omega^{HASH,SEQ}$
- $m6$: uses $\sigma^{QG,PAR}$ and ω^{RQ}

As expected, the number of hops decreases with an increasing degree of parallelism. Figures 8.17(a) and 8.17(b) indicate that the dominating performance factor is the chosen materialization operator. Whether we query q-grams on extraction level in parallel or in sequence has only minor impact. We can state that the range-based query plans perform surprisingly good. The number of replies is 1 for the strictly sequential variants $m1$ and $m3$. That it is also 1 for $m4$ indicates the high selectivity of the similarity predicate and a rather small range for the materialized attribute – most query plans at the peers in that range are dropped because of empty results. The selectivity of the predicate further influences the number of replies for the parallel q-gram operators. These variants and the range-based variant $m2$ produce several duplicated results. Figure 8.17(d) again shows that parallel approaches should be prioritized in dynamic and unreliable environments. The decreased availability for $m2$ is an abnormal outlier, but indicates that range queries can result in problems in specific settings, an issue that should be further investigated. Answer times, which are shown in Figure 8.18, are as expected and underline the benefits of parallel operators.

We further investigate the dependence of the q-gram operators on the queried distance. The larger the distance, the more effort is needed for these operators. Figure 8.19 shows that, in contrast, an increased distance does not influence the range-based processing. This time we used only similarity extractions. Thus, the queries are from class e. The characteristics of parallel processing result in more messages with more queried q-grams, but constant hop count. This does not hold for sequential q-gram operators, because the candidates are distributed over more peers, which results in longer sequences. The very small increase in the number of hops is not contrary to the cost model. The more candidates have to be queried, the higher the probability that this requires a larger number of intermediate P-Grid hops. Plots of the hops on M^2QP level show the expected results. Figure 8.19 shows that this is one of the rather seldom cases where there exists a drift between the differences in the number of messages and the differences



(a) Messages (b) Bandwidth

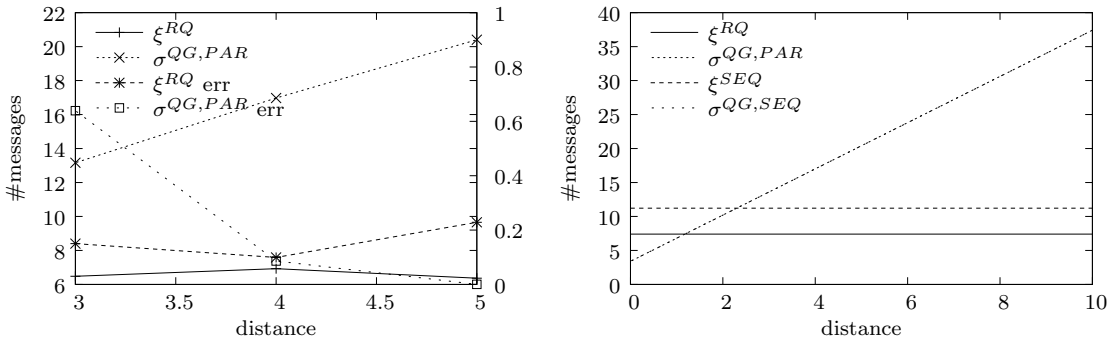
(c) Hops (d) Time for first reply

Figure 8.19: Similarity extraction for different distances

in the resulting bandwidth consumption. These experiments show that the benefits of q-gram operators decrease with increasing distances. Answer times show the same results as observed before for the comparison of sequential and parallel operators. Similar holds for the number of replies and the achieved robustness.

The q-gram operators for similarity extraction are the last single operators that we use to provide insights into the cost estimations. Again, we show absolute estimations and resulting relative errors for selected operator variants in Figure 8.20(a). As expected, the exists a certain distance from which on the q-gram operators get more expensive than the range-based operators. In the experimental results, this point is represented by a rather small distance. The actual trade-off depends on the used data and the characteristics of the network. In the results presented here, a rather small range of peers is responsible for extracted attributes. This makes the parallel range-based operator a meaningful alternative. A theoretical look on the scalability of several q-gram operators is provided in Figure 8.20(b).

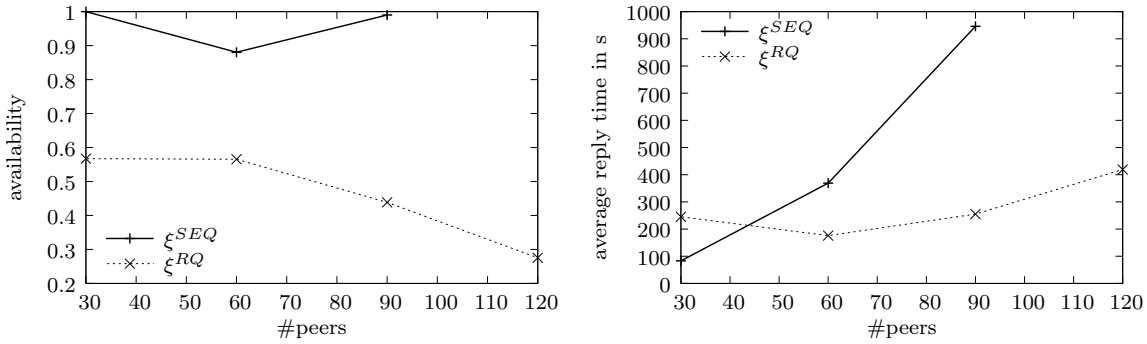
We omit any results for similarity materializations, because they behave like similarity extractions. Rather, we take a closer look on similarity queries on schema level. As argued before, range-based operators, all the same whether sequential or parallel, are not suited for this purpose, because they require to query for all triples in the system. We back this with the results shown in Figure 8.21. We ran a single unbound extraction from class e. On the first look, the availability of ξ^{SEQ} looks good, but no query at all was successfully finished for 120 peers. Moreover, this approach results in extremely high answer times. The range-based parallel approach performs bad as well, because there is too much load injected into the



(a) Real vs. estimated messages

(b) Scalability in messages

Figure 8.20: Cost comparison of similarity extractions



(a) Robustness

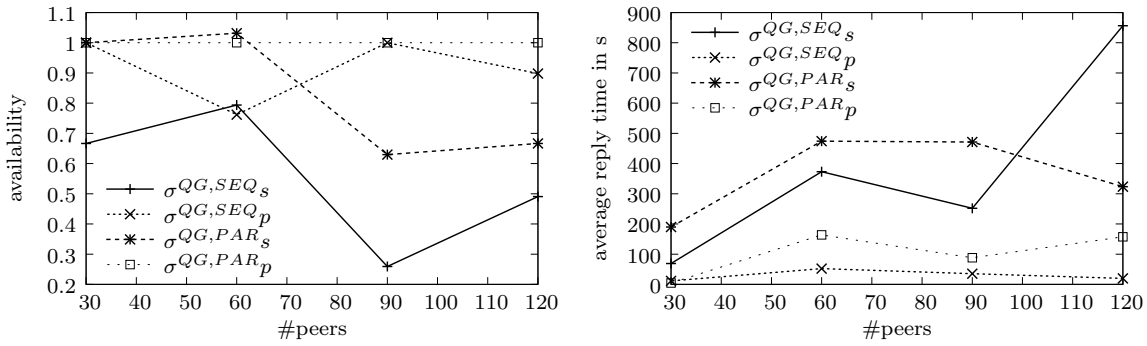
(b) Average reply time

Figure 8.21: Unbound extraction for similarity on schema level

system. This is especially an issue in the experiments presented here, because of the Planet-Lab characteristics. Availability and answer times for ξ^{RQ} illustrate this.

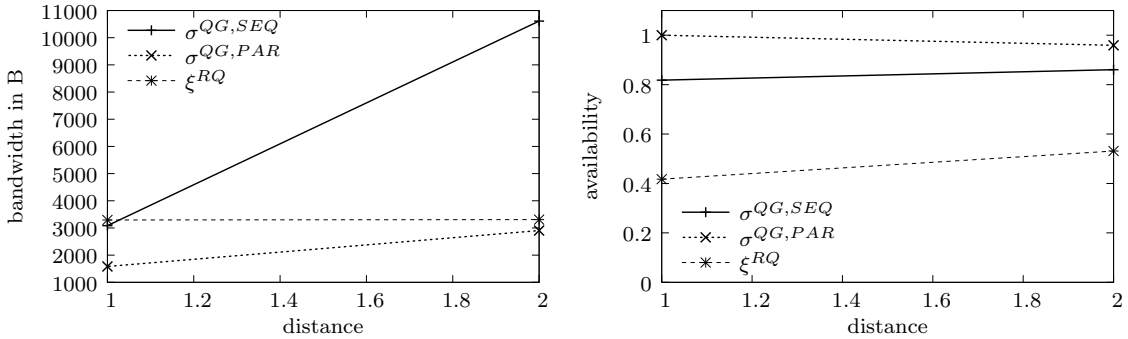
Thus, similarity on schema level is only feasible with q-gram operators. The differences in the number of messages, bandwidth consumption, and hop count are analog to similarity operators on instance level. This is intuitive, because there is no difference for the actual processing. More interesting are the availability and answer times. They are shown in Figure 8.22. The used queries are from class m. They contain a similarity extraction on schema level and two further materializations. The query identified by $\sigma^{QG,SEQ}s$ uses $\omega^{HASH,SEQ}$ and $\sigma^{QG,SEQ}p$ uses ω^{RQ}. In analogy, $\sigma^{QG,PAR}s$ uses $\omega^{HASH,SEQ}$ and $\sigma^{QG,PAR}$ uses ω^{RQ}. The strictly sequential variant $\sigma^{QG,SEQ}s$ shows poor availability, but the parallel approaches provide satisfying results. The one point above an availability of 1.0 is due to inconsistencies in the P-Grid network. This is very seldom, but can result in ambiguous answers. The average answer times in Figure 8.22(b) show the dominance of the materialization operator. All in all, similarity on schema level is an particularly expensive operation.

We finally illustrate this in Figure 8.23. Here, we used pure similarity extractions on schema level, conforming to class e. We again vary the queried distance. The sequential range-based approach is omitted. The linear increase in bandwidth consumption indicates the costs for such q-gram operators even for small distances. The plots for the number of messages and hop count



(a) Robustness (b) Average reply time

Figure 8.22: Similarity on schema level



(a) Bandwidth (b) Robustness

Figure 8.23: Schema similarity for different distances

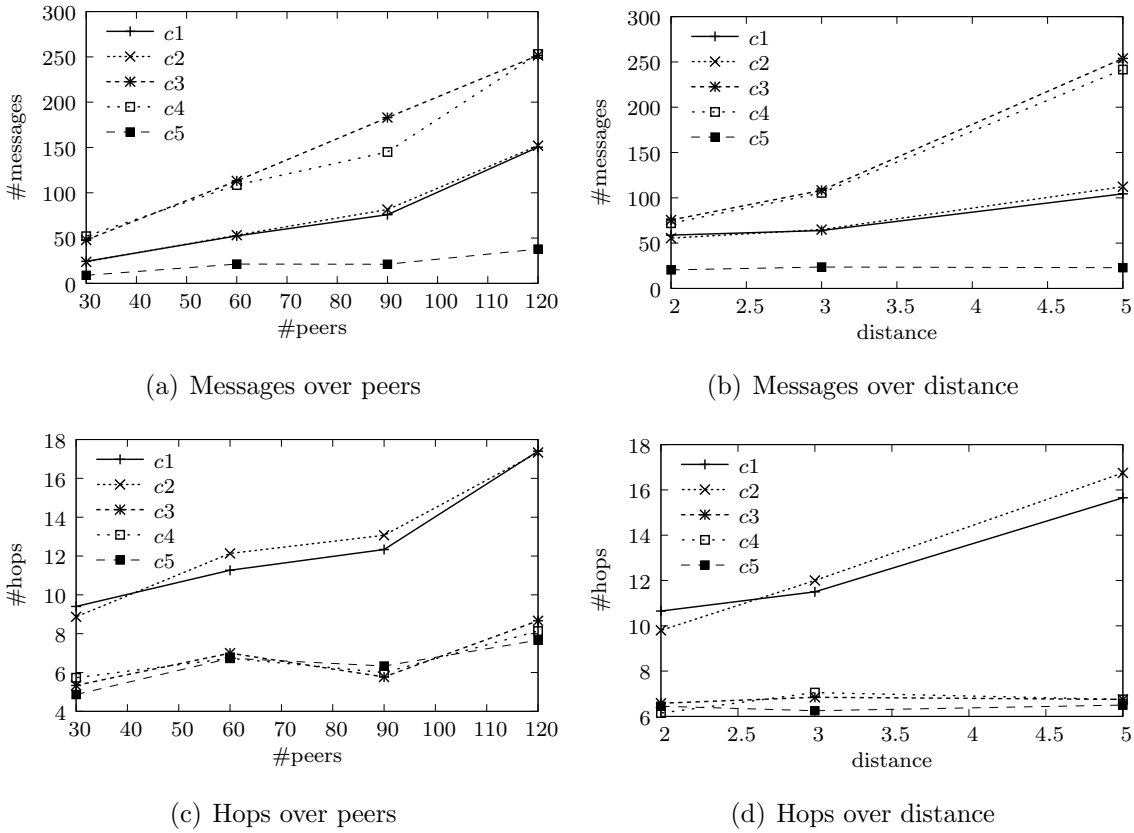
look analog. Despite the high costs, the q-gram operators still provide a satisfying robustness in terms of availability.

We were able to show that q-gram operators are a very promising alternative to other hash-based and also to range-based operators. Especially for supporting similarity on schema level they are very important. The trade-off between the benefit of and the effort for parallelism applies here as for other operators. But, sequential approaches are more meaningful here, because the resulting sequences are usually rather small. Nevertheless, basic extraction and materialization operators are suited to handle similarity on instance level as well. The results for other q-gram-based operators, such as prefix, suffix, and substring querying, look very similar to the results for similarity queries presented here.

Similarity Join Operators

The most complex similarity operators that we discussed are similarity joins. We evaluate different physical operators that support this operation next. The queries used for this are from class c. They contain two extractions and a similarity join on instance level. There are no subsequent materializations included. We included the following combinations in the tests:

- $c1$: uses $\bowtie^{HASH,PAR}$ ($\xi^{QG,SEQ}$) and ξ^{SEQ}



(a) Messages over peers

(b) Messages over distance

(c) Hops over peers

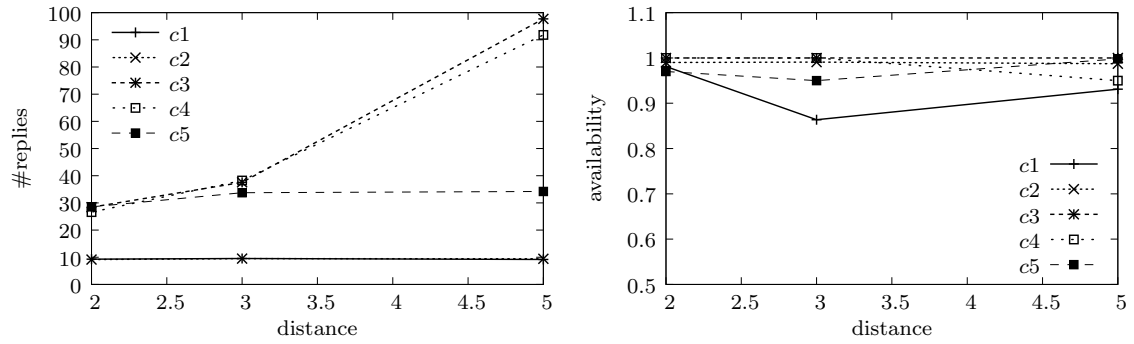
(d) Hops over distance

Figure 8.24: Similarity join operators

- $c2$: uses $\bowtie^{HASH,PAR}$ ($\xi^{QG,SEQ}$) and ξ^{RQ}
- $c3$: uses $\bowtie^{HASH,PAR}$ ($\xi^{QG,PAR}$) and ξ^{SEQ}
- $c4$: uses $\bowtie^{HASH,PAR}$ ($\xi^{QG,PAR}$) and ξ^{RQ}
- $c5$: uses $\bowtie^{LOC}$ and ξ^{RQ}

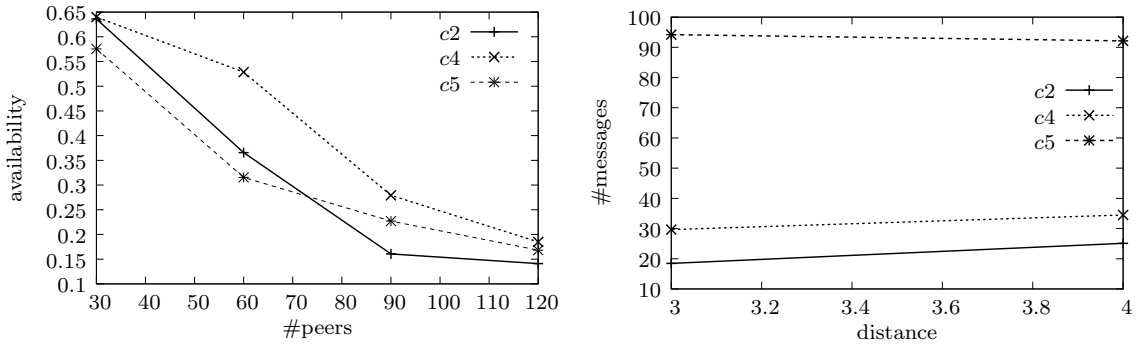
In Figure 8.24 we compare the resulting number of messages and hops for the different variants, with respect to the number of peers and the queried distance. The bandwidth consumption looks analog to the number of messages. The figures reveal that the driving performance factor in these queries is the join operation. The used extraction operators make almost no difference. The effort for all q-gram similarity joins scales linearly with the number of participating nodes and the queried distance. The hop count of the range-based approach is very close to the parallel q-gram approaches, but it requires significantly less messages. The hop counts perfectly reflect the achieved answer times.

The benefits of the parallel q-gram operators get evident when we take a look at the achieved robustness in Figure 8.25(b). Despite the range-based operators perform well on instance level, the availability achieved with parallel q-gram operators is still higher. The sequential operators bear the problems already discussed, but still perform satisfyingly. The increasing number of replies for the parallel operators (see Figure 8.25(a)) indicates that they produce more duplicate results with increasing query distance. This is due to the fact that more q-grams are used, which results in looking up identical candidate strings more often. This effect can be lowered



(a) Replies (b) Robustness

Figure 8.25: Reply count and robustness for similarity joins

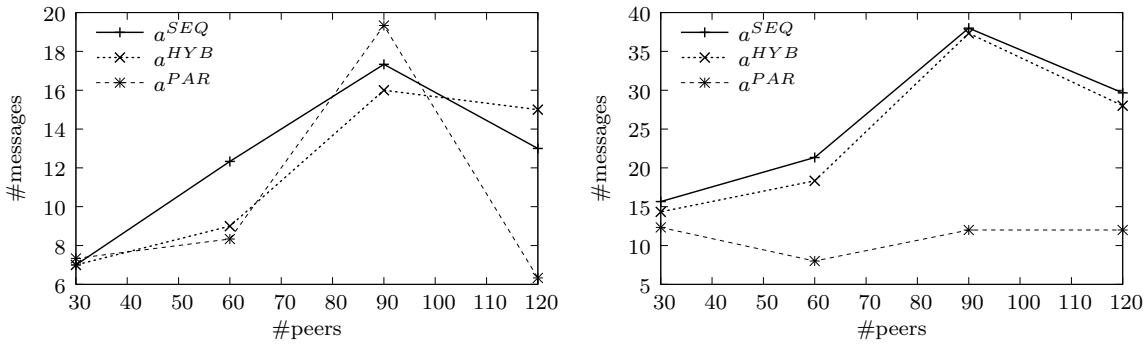


(a) Robustness (b) Messages

Figure 8.26: Schema similarity joins with unbound extraction

by querying q-grams in sequence, as the results for $c4$ show. Note that this is not always the case. As a consequence, even the range-based processing of similarity joins is a feasible option.

The results for similarity joins on schema level correspond to the results presented above, as long as one of the input sides is bound, at least by a similarity extraction. The advantage of q-gram-based similarity joins is that they do not require the second side to be bound – a binding results from the hash-based approach. We finally show that similarity joins perform bad as well if none of the input sides is bound. Again, the sequential approach is omitted, because it would require to contact all peers in sequence. The parallel range-based approach requires a two-phase range querying, each time involving all key space partitions. An alternative is to use explicit synchronization or hash-based join operators. All the alternatives are impractical. We exemplary show this in Figure 8.26 by plotting the availability for parallel range-based and q-gram approaches. The used operator combinations correspond to those from the last test. This time, the join predicate involves similarity not only on the instance values, but also on the attribute names. The shown availability is very poor for all variants. Due to the corrupt results, the statistics for performance measures are misleading – remember that statistics are collected only from received replies. But, Figure 8.26(b) illustrates the constant overhead of the range-based approach and that the number of messages for q-gram joins increases with an increasing distance.



(a) Messages for limit 5

(b) Messages for limit 20

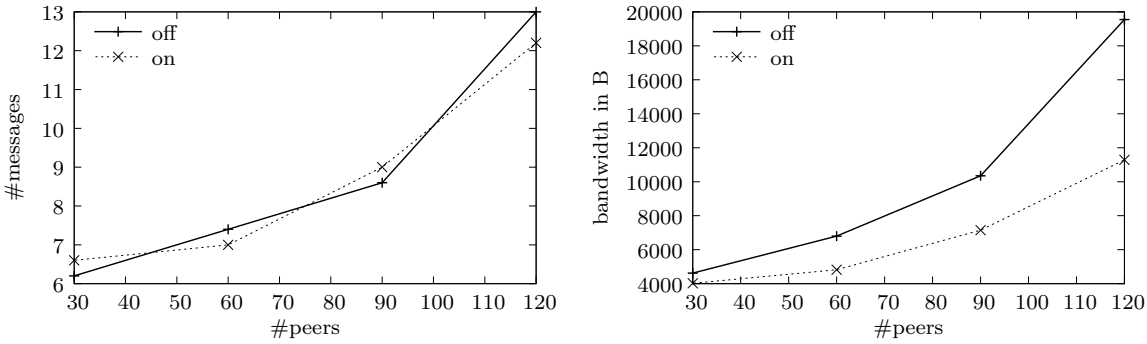
Figure 8.27: Effect of different limits for φ^{SYNC}

Similar to q-gram operators for extraction and materialization, q-gram-based similarity joins are a very valuable. They result in a noticeable overhead, but leverage robustness and performance of query processing significantly. In limits, they are also suited to process similarity joins in schema level. However, schema similarity needs at least one binding to be supported in a satisfyingly efficient and accurate manner. Numerical similarity is covered by the top-N operator, which is briefly summarized in the next section.

Ranking Operators

We omit an evaluation of the proposed sophisticated ranking operators. These operators show a wide range of facets and performance factors. Thus, they demand a stand-alone evaluation. The general principle of the range-based top-N operators can be found in [210]. The main performance factors are the size of the involved interval and the number of re-initiated rounds.

A very detailed evaluation of skyline operators is presented in [142]. The main performance factors for these operators are identified in the number of peers in the range that receive subskyline queries. A further issue is the robustness of operators, which is leveraged with an increased degree of parallelism. [142] also underlines the importance of the online and evolving extensions for unreliable and dynamic environments.



(a) Messages

(b) Bandwidth

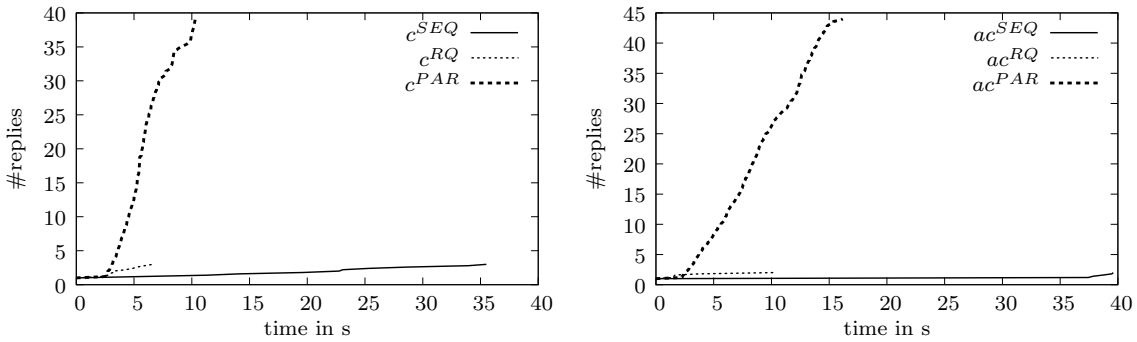
Figure 8.28: Tuple pipelining

We use ranking queries only to show the benefit of explicit synchronization, which can help to reduce the number of generated messages. The following results can be used to deduce the general benefits of synchronizing operators. Figure 8.27 shows that the provided limit does not influence the scalability in terms of messages for sequential operators. We used queries of the class a. Query a^{SEQ} uses a sequential extraction ξ^{SEQ} and a subsequent sequential materialization $\omega^{HASH,SEQ}$. This implies a local ranking operator φ^{LOC}. a^{HYB} uses a parallel extraction ξ^{RQ} and a sequential materialization $\omega^{HASH,SEQ}$, which requires a synchronizing ranking operator φ^{SYNC}. a^{PAR} uses parallel range queries in the extraction and the materialization. This also requires φ^{SYNC}, but results in more input messages for this operator. Consequently, this can reduce the overhead compared to the approaches with sequential parts. The collection of intermediate results at one designated synchronization peer results in less updated revisions, which in turn results in less sent reply messages. This provides first insights into the benefits of early synchronization.

8.2.4 Further Issues of Query Plan Processing

Finally, we take a brief look on some further issues of query plan processing and the performance of general processing strategies.

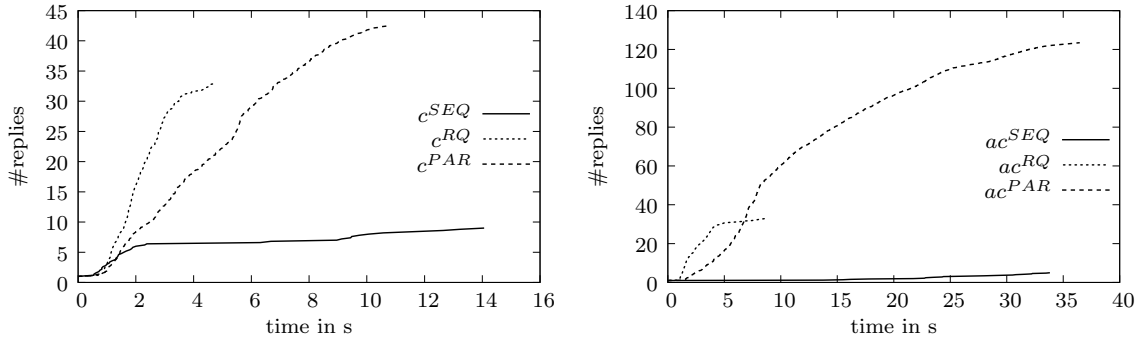
Figure 8.28 shows effect of tuple pipelining using a simple extraction query from class e. Enabled tuple pipelining (indicated by `on` in the figure) results in the same number of messages as disabled tuple pipelining, but significantly less bandwidth consumption. This backs what we discussed before. Peer pipelining behaves like a mixture between parallel processing and sequential processing: it also bears the advantage of low answer times, but also the risks of sequential processing approaches concerning robustness. We omit evaluation details, because they do not provide any new insights.



(a) c (b) ac

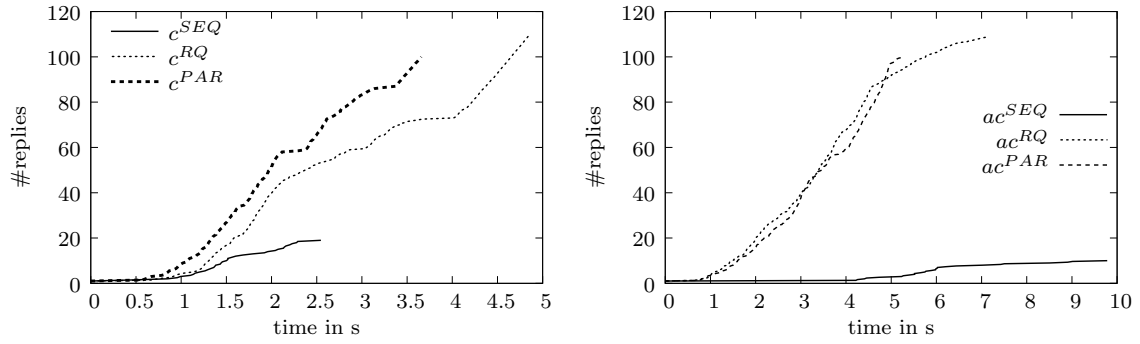
Figure 8.29: 25 peers

Next we provide some meaningful insights into the effects of sequential and parallel operators by choosing a different illustration for experimental results. In Figures 8.29 to 8.31, we plot the number of replies received over time. The total number of received replies reflects the degree of parallelism and thus, the total number of generated messages. The rise in the plots indicates the minimal and average answer times. We present results for queries from classes c and ac run in the local setup on the DBPedia and Mondial data. All queries use ξ^{RQ} for extraction on leaf level. For remaining materializations, they apply $\omega^{HASH,SEQ}$ (in c^{SEQ} and ac^{SEQ}), ω^{RQ} (in c^{RQ} and ac^{RQ}), and $\omega^{HASH,PAR}$ (in c^{PAR} and ac^{PAR}) respectively. By this, we varied



(a) c (b) ac

Figure 8.30: 50 peers

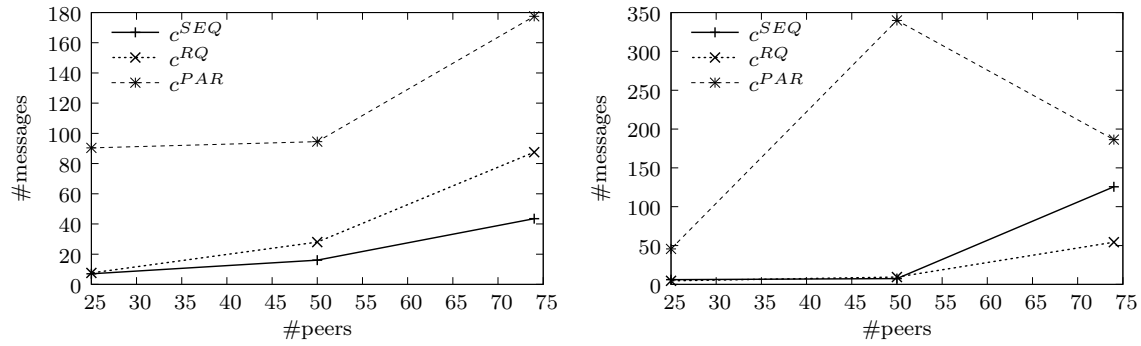


(a) c (b) ac

Figure 8.31: 74 peers

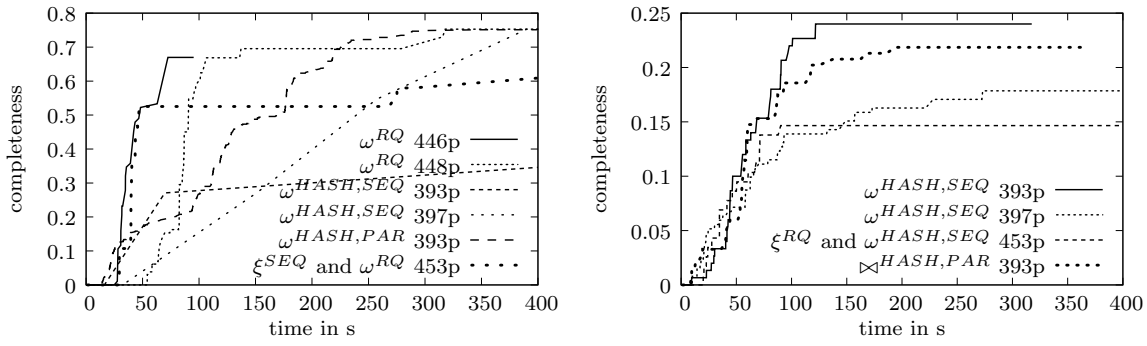
the general degree of parallelism in the query plans. All queries were run successfully. Thus, the last reply indicated represents the point in time where an availability of 1.0 was reached. The figures show that sequential operators produce the same result with less but later replies compared to parallel operators. The parallel hash-based materialization can result in a very high amount of messages and replies.

The figures also show that the performance of the parallel approaches depends on the actual query and the network size. With increasing peer count, the parallel range-based operators may perform worse than the parallel hash-based operators. This is because ranges grow with increasing peer count. But, this cannot be assumed as a general fact. In cases, even a sequential processing strategy can result in fast replies. Also, results from parallel processing strategies may be received later than all replies resulting from a more sequential approach. It is interesting to observe that reply times decrease with increasing number of peers. In the experiments we kept the size of inserted data constant. Thus, with increasing peer counts, each peer is responsible for less index items. The observed decrease in response times can only be due to lowered costs for local processing of queries. Even if it was not planned, this evaluates the benefit of distribution in comparison to local processing. Of course, this is only one example, but indicates the general relation. To provide an accurate understanding of the performance of all



(a) c (b) ac

Figure 8.32: Messages



(a) m (b) c

Figure 8.33: Full-scale Planet-Lab experiments

three query variants, we additionally show the number of received replies with respect to the network size in Figure 8.32.

These results underline the observations we made for the trade-offs between sequential and parallel processing. Further, they provide interesting insights into the relation between range-based and hash-based operators. The need of an appropriate cost model for enabling scalable and reliable query processing is observed here again.

Before concluding the evaluation of physical operators and processing strategies, we show that the few results gained in the very unreliable full-scale Planet-Lab tests look similar. In these tests, we could only finish rather simple queries. In Figure 8.33(a) we illustrate successful executions of queries from class m. The depicted queries differ in the used materialization operators. We indicate the used operators in the figure, together with the network size. We choose a similar illustration as before, but this time we plot the availability over time rather than the number of received replies. Like this, we indicate performance as well as the usually poor availability in this tough setup. The number of replies (and thus, messages) can be deduced from the number of steps in each plot. The smoother a plot is, the less replies were received. In Figure 8.33(b) we accordingly illustrate successful executions of queries from class c.

Finally, we illustrate the number of replies over time for some q-gram queries from class c run in Planet-Lab. Figure 8.34 underlines the differences between the q-gram operators and

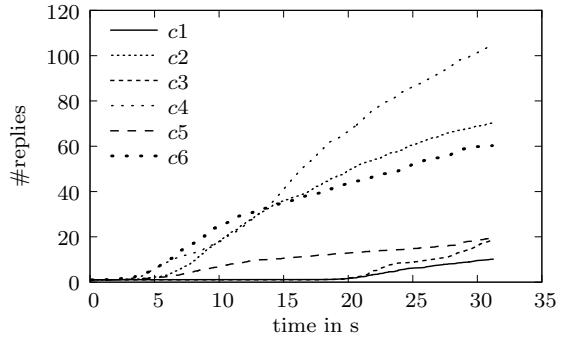


Figure 8.34: Q-gram queries with 384 Planet-Lab peers

shows that a significant amount of result data is already received after a relatively short time. The used queries include the following combinations of operators:

- $c1$: $\bowtie^{HASH,PAR}$ ($\xi^{QG,SEQ}$) and $\sigma^{QG,SEQ}$
- $c2$: $\bowtie^{HASH,PAR}$ ($\xi^{QG,SEQ}$) and $\sigma^{QG,PAR}$
- $c3$: $\bowtie^{HASH,PAR}$ ($\xi^{QG,PAR}$) and $\sigma^{QG,SEQ}$
- $c4$: $\bowtie^{HASH,PAR}$ ($\xi^{QG,PAR}$) and $\sigma^{QG,PAR}$
- $c5$: $\bowtie^{HASH,PAR}$ ($\xi^{QG,SEQ}$)
- $c6$: $\bowtie^{HASH,PAR}$ ($\xi^{QG,PAR}$)

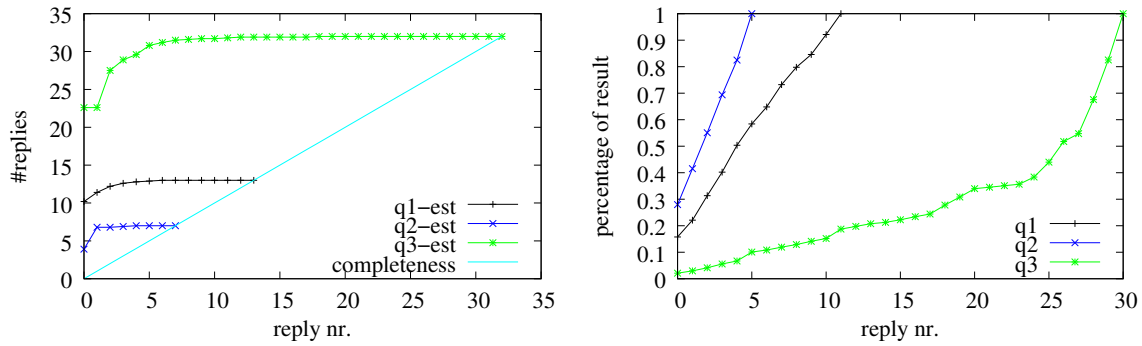
The last presented results confirm that the proposed concepts are also suited for query processing in extremely dynamic and unreliable environments. If the sophisticated operators perform as expected in the tough Planet-Lab setup, they are a valuable basis for enabling Public Data Management. We expect according environments to show a mixture of the characteristics of the reliable local and the very unreliable Planet-Lab setup.

8.2.5 Query Completeness

Completeness estimation as proposed in Chapter 7 relies on correct estimations for all applied processing techniques. This involves an appropriate technique for range queries, which are an essential ingredient of the framework. Thus, we first evaluate the approach for estimating range queries in P-Grid presented in Section 7.2.1. Based on this, we focus on the evaluation of completeness estimation for complex query plans involving all introduced routing methods.

Completeness Estimation for Range Queries

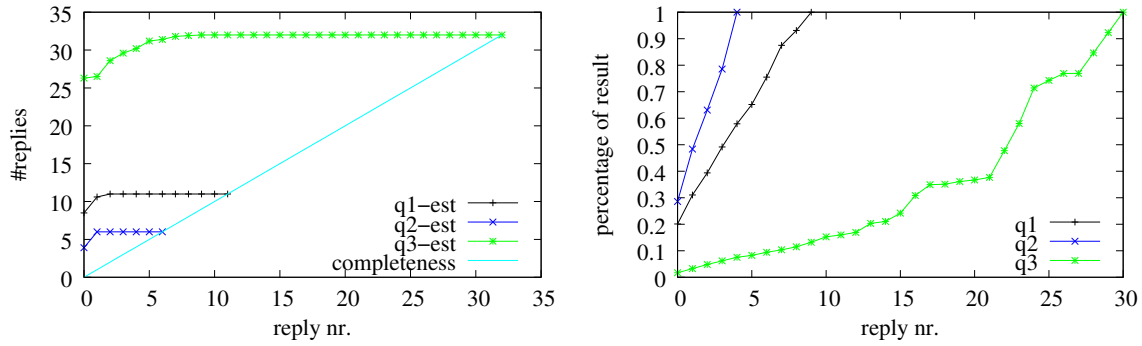
The following evaluation focuses on the completeness estimation of range queries. We assume that P-Grid guarantees the availability of at least one peer per key space partition even in very dynamic or unreliable setups like Planet-Lab. The experiments were run on the local reliable setup with up to 61 nodes. Parts of the results have already been presented in [109].
For these tests, we used a the Imdb movie data. In the local setup with a replication factor of 1 and the three standard indexes from Section 4.2.1, these data resulted in a P-Grid tree with an average number of leafs, maximal path length, and average path length of 32, 8, and 5



(a) Estimated number of replies with corrections (b) Completeness on data level

Figure 8.35: Replication:1, maxref:1

respectively (for a). For a factor of 2, the values were 19, 6, and 4.5 respectively. As expected for skewed data, the resulting P-Grid tree was not balanced. Almost 40% of the leafs were located under key prefix 0 and the tree was deeper and wider under key prefix 1.

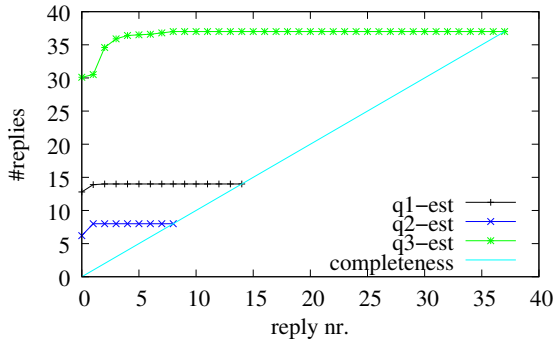


(a) Estimated number of replies with corrections (b) Completeness on data level

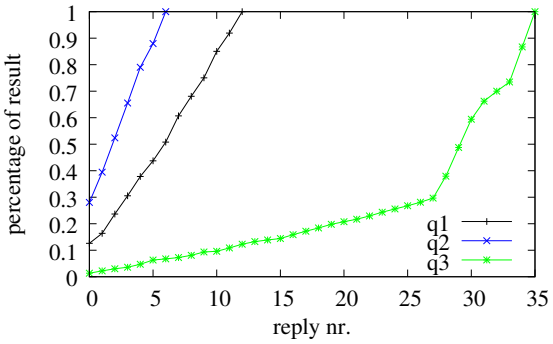
Figure 8.36: Replication:1, maxref:3

In order to evaluate the influence of the number of references for one level of the local routing table we built three environments, using a maximal number of references of 1, 3, and 5. As this part of the evaluation focuses on pure P-Grid range queries, we only ran queries of class e from Figure 8.5 (Section 8.2.1, page 182). A query mix of three different range queries, involving different parts of the trie and therefore resulting in a different number of replies, was run. Query $q1$ asks for all index entries with prefix 10 (range 100...0-101...1), $q2$ for all with prefix 11. Due to the reliability and robustness of the local setup, we were able to even run an unbound query: $q3$ represents a range query that involves the whole P-Grid tree. Even if such a query would be rejected, it allows for evaluating the completeness estimation for the most challenging case. Each query was initiated 10 times, each time on a randomly chosen node. In the following, we present and discuss the results of the described experiments.

Figures 8.37(a) to 8.40(a) show the number of replies we estimated using the proposed technique with each reply received. Additionally, the straight line represents the actual completeness on peer level. The figures clearly show that our method always estimates the number



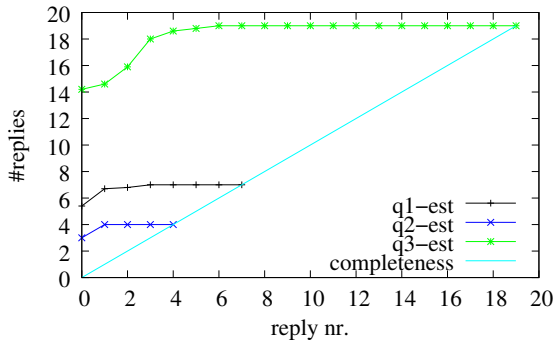
(a) Estimated number of replies with corrections



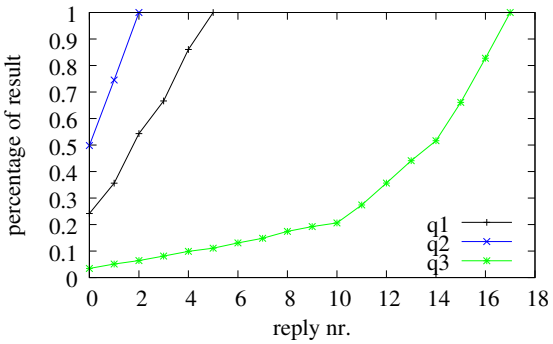
(b) Completeness on data level

Figure 8.37: Replication:1, maxref:5

of replies correctly at the end. Moreover, it gets evident that only a small number of first replies is needed in order to determine a correct value in the end. As expected, the higher the number of references for each level of the routing table, the more exact the initial estimation and the less corrections are needed. The figures also show that in this case the size of the temporary errors is smaller than with lower references per level. The differences in the number of replies for identical queries are due to the need for starting networks with different parameters from scratch every time. By this, and the application of a random-walk strategy in order to build the P-Grid trie, this results in, only slightly, different overlay trees.



(a) Estimated number of replies with corrections

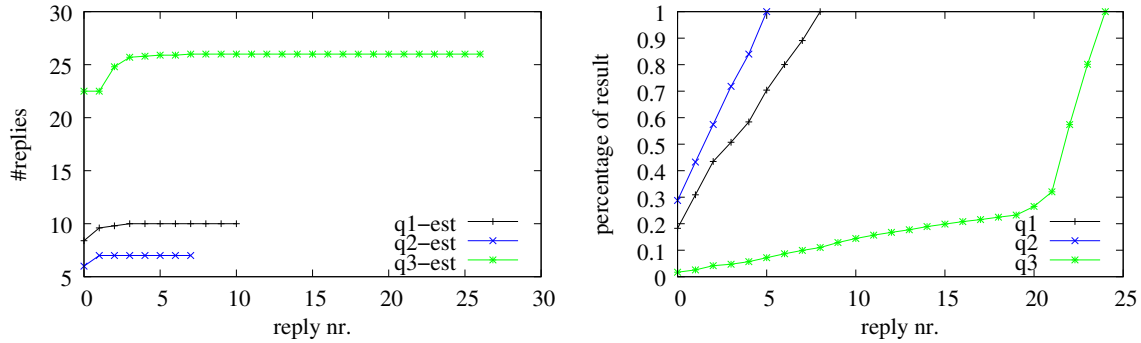


(b) Completeness on data level

Figure 8.38: Replication:2, maxref:1

Figures 8.37(b) to 8.40(b) we additionally show the correspondence between completeness on data level and on peer level. The figures show the percentage of the final result received with respect to the number of replies received. Since we use the number of replies on the x axis, a perfect mapping between data and peer level would be indicated by a straight line. The plots show that especially for the two queries resulting in less answers the development of the result size is almost linear. For the query involving the whole P-Grid trie the last query replies contain larger fractions of the result than earlier replies. Even if P-Grid implements a sophisticated load-balancing, there might exist keys a particular high number of data items is mapped to. P-Grid's load-balancing technique splits high frequented key space partitions

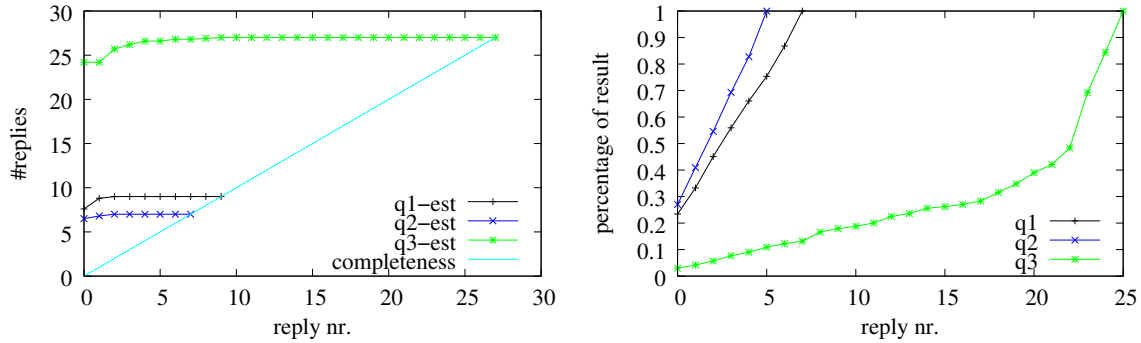
more fine-granular than others, but does not "split" single keys. Thus, some peers are still responsible for a higher number of items than others. Due to the locally used storage system, the answer time correlates to the amount of data to be processed locally. Therefore, replies from these peers arrive with later replies, resulting in a higher increase of the result size with the final answers. The figures show that the mapping from completeness on data level to the completeness on reply level is satisfyingly realistic in load-balanced overlay systems.



(a) Estimated number of replies with corrections (b) Completeness on data level

Figure 8.39: Replication:2, maxref:3

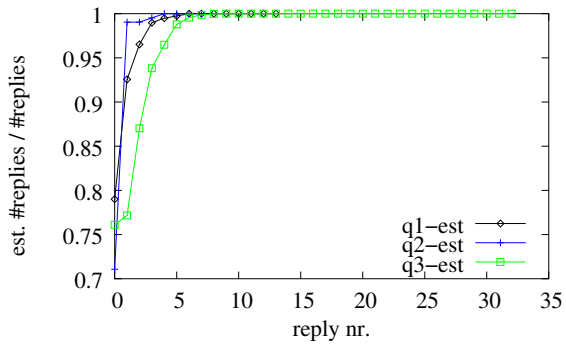
The smaller the part of the trie involved into range query processing the less information is needed in order to achieve exact estimations. For the first two queries, even the settings using a replication factor of 1 and/or a maximal number of references per level of 3 and 1 are quite satisfying. As sub-trees are queried more probably than the whole tree, this shows that the proposed method provides quick and exact estimations even with low information. This also indicates the effectiveness for larger scaled and unreliable systems where, in turn, more information shall be contained in the local routing tables.



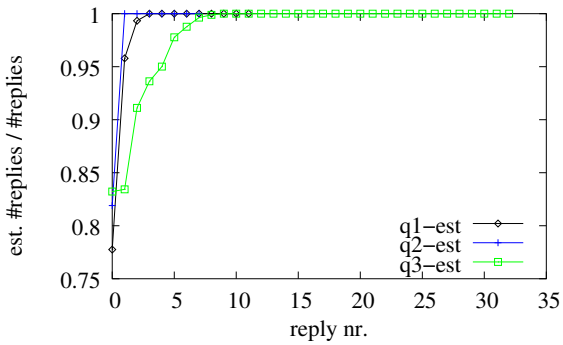
(a) Estimated number of replies with corrections (b) Completeness on data level

Figure 8.40: Replication:2, maxref:5

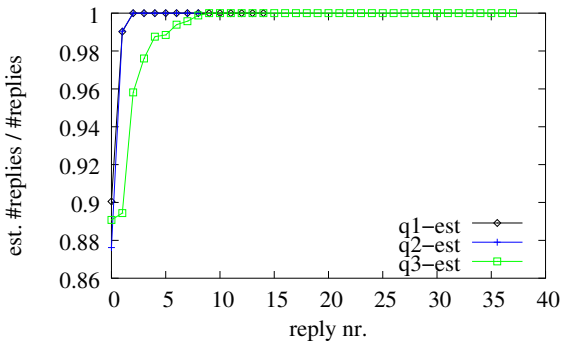
Summarizing, we can state that for each of the considered cases we only need a fraction of replies in order to achieve an exact completeness estimation. As the method goes along with very low additional effort, this proves its powerfulness for trie structured overlays in general.



(a) maxref:1



(b) maxref:3



(c) maxref:5

Figure 8.41: Estimated relative number of replies with corrections

The last figures show the relative completeness ($\frac{\text{estimated \#replies}}{\text{final \#replies}}$) estimated with each reply. Thus, it illustrates the ratio of error correction. Moreover, this time the queries were run on two different networks for each setting, each of them run for a different time before starting queries. Results from the hence four runs were averaged. Thus, effects of slightly different overlay tries are eliminated. Figure 8.41 shows that the ratio of correction is always almost equal for each of the used environments. Following, independent from the query actually initiated, completeness estimation is comparably good and corrections provide equally good improvements with respect to the size of the final result. The figures also show that the initial estimate is good for all tests, but it is better if more references are stored at each routing table level. As expected, an error correction can be recognized only for the first query replies and converges to 0 for later replies. Another important observation is that the estimation for the queries with less replies are very exact with little information and that the corresponding plots approach each other with rising numbers of references.

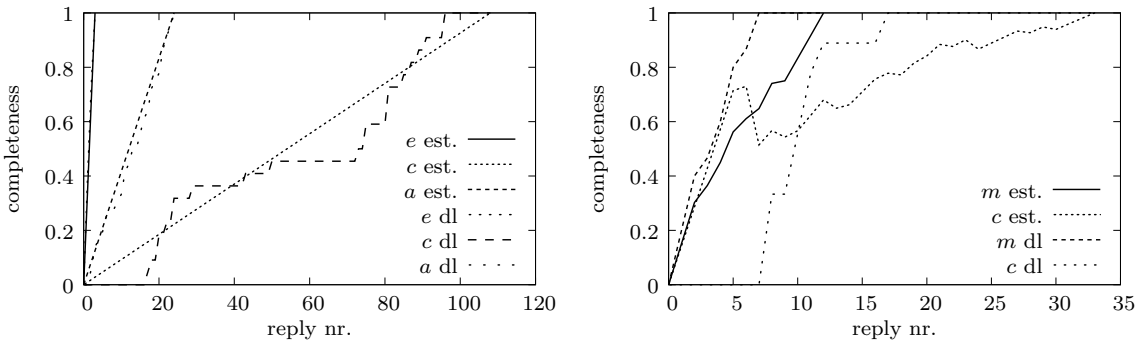
All in all the proposed method for completeness estimation is absolutely satisfying. The initial estimation, based on no further knowledge than the local one, is quite good for any type of query and environment. Even if this first estimate is erroneous, only a small amount of replies is needed in order to determine an exact estimate.

In the presented experiments, we omitted any dynamics, which is of course a natural ingredient of P2P systems. Providing complete results under churn is a task the overlay is responsible for, not the system layered on top of it, such as UniStore. If the overlay guarantees complete results, which modern overlay systems achieve by applying adequate replication algorithms,

the proposed approach will work correct, as shown above. Moreover, in recent time the super-peer approach, relying on a fraction of all peers which are rather reliable and robust, gains boosted attention. Referring to this, the static network used in our tests corresponds to such robust super-peers. The only point where minor errors may occur is if a peer crashes right after receiving and acknowledging a query, but before replying to it, which is rather unlikely.

We include the consideration of more dynamic networks in the following evaluation of the whole process of completeness estimation. By this, we evaluate the robustness of the used overlay in combination with completeness estimation for all operators supported by UniStore.

Completeness Estimation for Complex Queries



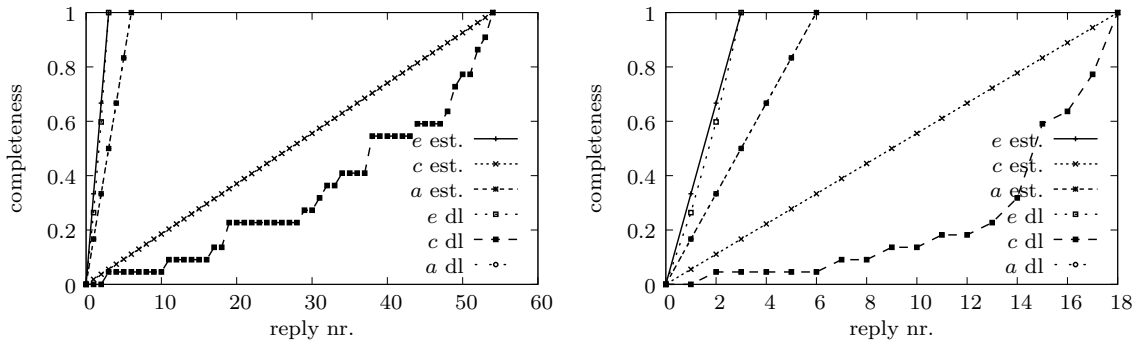
(a) Only DOR routings

(b) Only FNR routings

Figure 8.42: Accuracy for separated routing methods

In this section, we evaluate the proposed approach for completeness estimation running an extended set of experiments on the local environment. The purpose is to proof the accuracy of the techniques and to discuss special issues. We ran up to 74 independent instances in the local setup. Further, we repeated these tests on Planet-Lab setups. At the time of our experiments, we could allocate only very few resources in Planet-Lab, so that we were able to run only a couple of our queries in a meaningful sense. In each run we built a P-Grid network with standard parameters from scratch. We based our experiments on the mixture of triple data from DBPedia, the geographical data in relational format from Mondial, and a small set of ontology data. Parts of the following results have already been presented in [105].

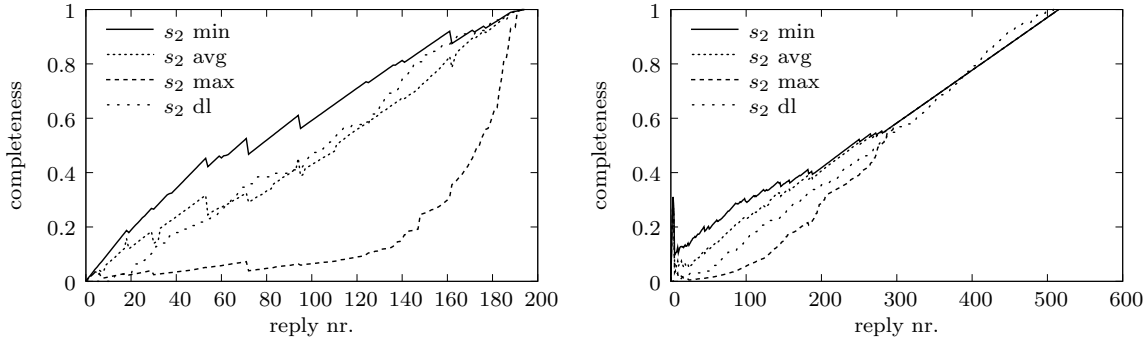
After a certain waiting time for establishing a suitable overlay trie, we initiated queries conforming to the classes introduced in Figure 8.5 (Section 8.2.1, page 182). We built two different indexes, which resulted in a total of about 16,000 index entries. As described before, the set of all generated keys shows a skewed heavy-tail distribution (power-law like). We used a value of 5 for P-Grid's replication factor as well as the maximal number of references per routing level. The average number of leafs, maximal path length and the average path length were 35, 7 and 5 respectively. The average number of single entries each peer was responsible for was approximately 5,000. We ran the tests on networks of size 25, 50, and 74. For the Planet-Lab tests, we gathered as many nodes as possible. As shown in Section 8.2.4, this results in poor performance and completeness for specific queries. Since we focus on the quality of completeness estimation, this is fine as a counterpart to the stable local environment. In the tests, we were able to include about 400 nodes in each run.



(a) DOR with one sequential operator

(b) Only sequential operators

Figure 8.43: Sequential operators



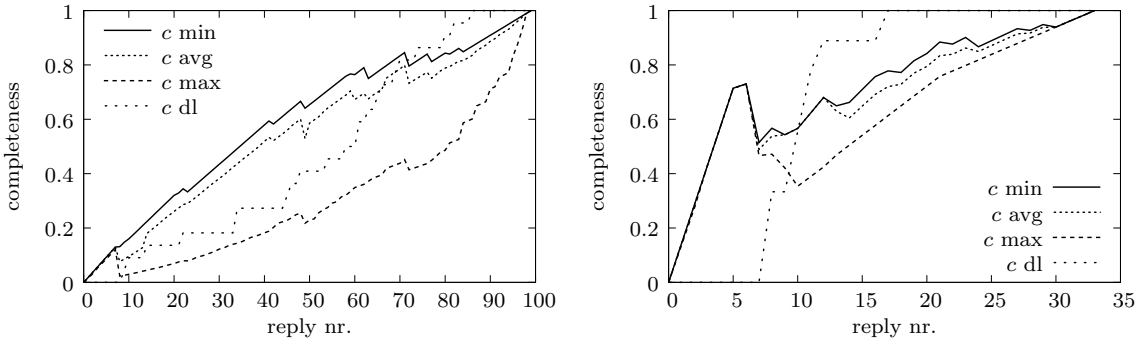
(a) On 74 peers

(b) On 50 peers

Figure 8.44: Minimal, maximal, and average estimation for s_2

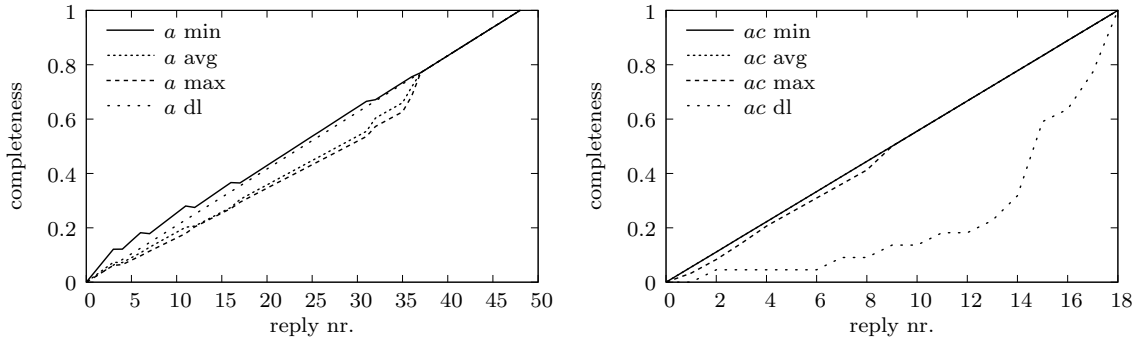
The aim of the evaluation is to show that the proposed method works correctly. Further, we will highlight the benefits of using MiMes for achieving a higher accuracy in early query states and compare the three different estimation techniques min, max and avg. From all experiments, we selected the results that are particularly suited for this. In most of the following figures we plot the number of received replies on the x axis. On the y axis we use the estimated completeness at the time of the reply. Thus, the actual completeness on peer level for each query is the straight line from $(0, 0)$ to $(\rho, 1)$. We omit this line in order to improve readability. Rather, we plot completeness on data level (denoted by dl) to additionally show the correlation between both levels. Each query was initiated 3 times, each time on a randomly chosen node. The following figures are created using the averaged values for each type of query. If not stated otherwise, the presented results are gathered from the biggest local network that still provided meaningful results, the 74 peer network.

We begin with evaluating the accuracy of completeness estimation separately for the supported routing methods. As already shown in Section 8.2.5, DOR routings can be estimated very accurately. To briefly show that this also holds for the setup used here, Figure 8.42(a) illustrates that completeness is estimated very accurately with first replies. For this, we used only range-based operators like ξ^{RQ} and ω^{RQ} in the query plans.



(a) With $\omega^{HASH,PAR}$ (b) With $\omega^{QG,PAR}$

Figure 8.45: Minimal, maximal, and average estimation for c

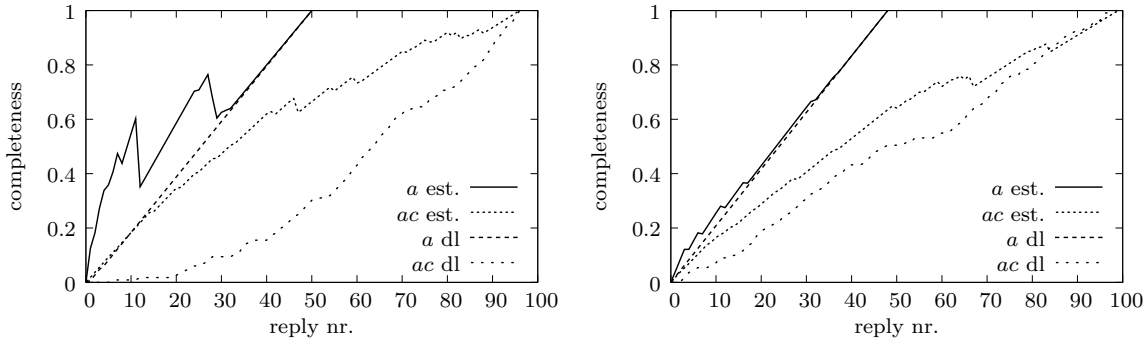


(a) a (b) ac

Figure 8.46: Minimal, maximal, and average estimation for a and ac

A more challenging task is to satisfyingly estimates FNR routings, because more information from the routing graph is needed for that. Figure 8.42(b) shows representative results by plotting the value of the minimal estimation calculated with each received reply for strict usage of FNR. Sequential operators like ξ^{SEQ} belong to the class of FNR operators, but as shown before they perform rather bad and should be avoided on leaf level. Moreover, they do not represent a specific challenge for estimating query completeness. Thus, we use only q-gram-based operators in the query plans used for evaluating FNR routings. To enable this, we used similarity selections and joins.
If only FNR is applied (8.42(b)), there is a clear gap between estimated and actual query completeness. Depending on the number of FNR parts, estimation gets inaccurate in early query states. This gap can also be observed with less FNR parts, e.g., when integrating sequential operators (8.43(a)). We illustrate this in Figure 8.43, in which we vary the fraction of sequential operators. All figures so far show that completeness on data level is satisfyingly good approximated by peer level, albeit the drift between both increases with rising query complexity.

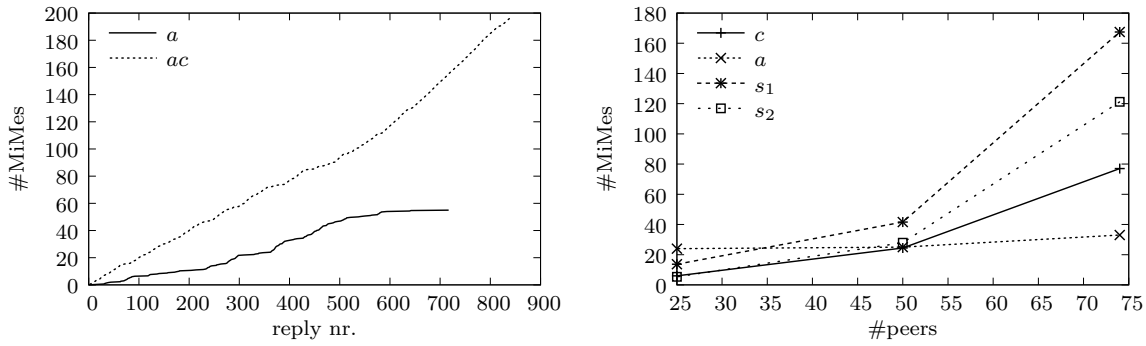
Next, we investigate the impact of the introduced average and maximal estimation methods. As DOR routings seem to be a by far lower challenge than FNR routings, we focus on queries containing at least one FNR routing. In Figures 8.44 to 8.46 we show corresponding results for selected queries. The three methods can be very close, but there can also be significant



(a) High parallelism, no MiMes

(b) High parallelism, MiMes

Figure 8.47: Impact of MiMes



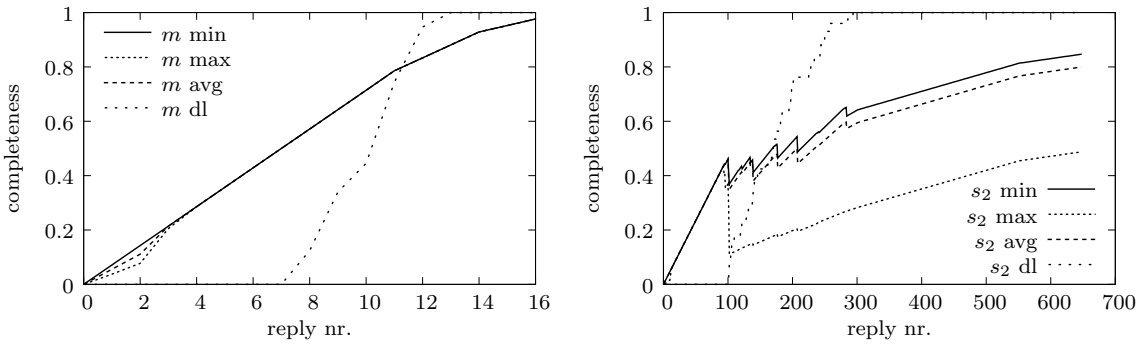
(a) Extra messages over time

(b) Extra messages vs. network size

Figure 8.48: Extra effort for MiMes

differences. As expected, the minimal estimation is always above the actual completeness, whereas the maximal one is mostly located below. To our pleasure, completeness on data level is mostly very well approximated by both, minimal and average completeness. The maximal method is rather pessimistic, resulting in a predicted completeness far below the actual one – but this also means, it provides what it was invented for: a guaranteed lower bound. The observed probabilistic guarantee approaches 90% for the maximal estimation rather quickly, whereas the average one usually balances between 50%-80%. As expected, the proposed guarantee is around 2%-5% for the minimal estimation technique. Figure 8.46(a) shows that the minimal estimate may even be the most accurate one. Figure 8.44(b) (one result from a run on 50 peers) illustrates that the point where all three methods meet each other (indicating the point of a 100% guarantee) can be reached even with 50% of all replies received. Figure 8.46(b) again shows the pessimistic character of the maximal estimation and the occasionally good quality of the minimal technique.

All the results presented up to here were gathered using MiMe support. We believe in the small overhead worth for achieving a higher quality of estimation. This is approved by the results shown in Figure 8.47. We chose to picture a query with high degree of parallelism, because MiMes become particularly advantageous in this case. The differences between 8.47(a) and 8.47(b) reveal that without MiMes corrections to the estimated value are bigger and occur



(a) m (b) s_2

Figure 8.49: Queries on Planet-Lab

more often. The overhead paid for this increases linearly with the number of received replies (8.48(a)). In Figure 8.48(b) we plot the number of cumulated MiMes for selected queries vs. the network size. As approximated before, this number increases linearly. Small deviations are due to queries initiated on different peers and a slightly varying amount of input data.

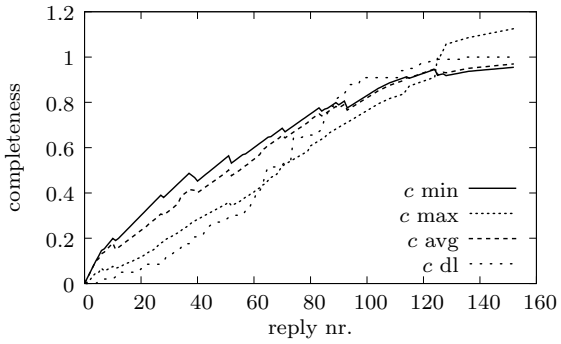
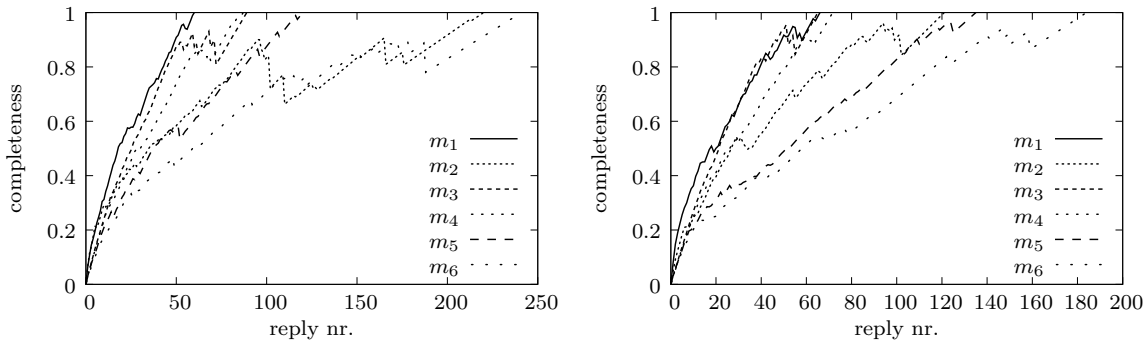


Figure 8.50: Errors in c on 393 peers

Next, we show that scalability in terms of network size and failure rates is no issue for the achieved accuracy by getting back to some successful Planet-Lab experiments. In Figure 8.49 we picture the completeness plots for two selected queries. All three estimation techniques behave analog to the local setup. As seen before, completeness on peer level is estimated very exactly. This matches the data level very well in most cases, but obviously not in all situations. An irritating point is that our maximal completeness estimation reaches a value higher than 1.0 in Figure 8.50, which should not happen by implementation. This can only be due to high inconsistencies in the overlay structure, caused by the painfully slow nature of Planet-Lab. Currently, we are investigating this issue in detail. In some earlier experiments, we ran some queries on the Imdb data and gathered statistics of completeness estimation. These results, which are shown in Figure 8.51, show that our approach also works for different data sets.

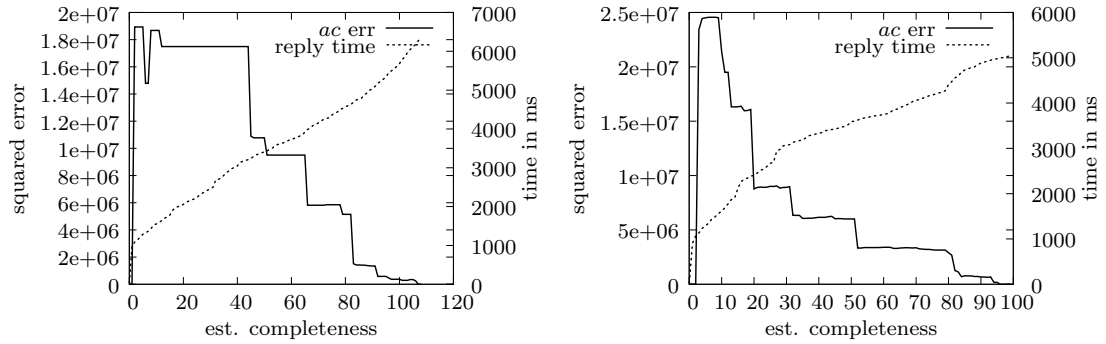
Last but not least, we present a small example how completeness estimation is applied internally in UniStore. Again, we ran two different plans for the aggregation query *ac*, but this time we stopped processing after a certain completeness was reached. We compared the calculated result against the expected one and measured the benefit we gained in terms of reply



(a) With MiMes (b) Without MiMes

Figure 8.51: Planet-Lab queries on Imdb data

time. The results are presented in Figure 8.52. As expected, the error between calculated and expected aggregate decreases with increasing input. But, the figure shows, that this can be satisfyingly recognized using the proposed method. Here we only apply completeness estimation on a single aggregating peer – this should get even more attractive when using it over different levels of according spanning trees in order to find a good point in time to determine partial aggregates.

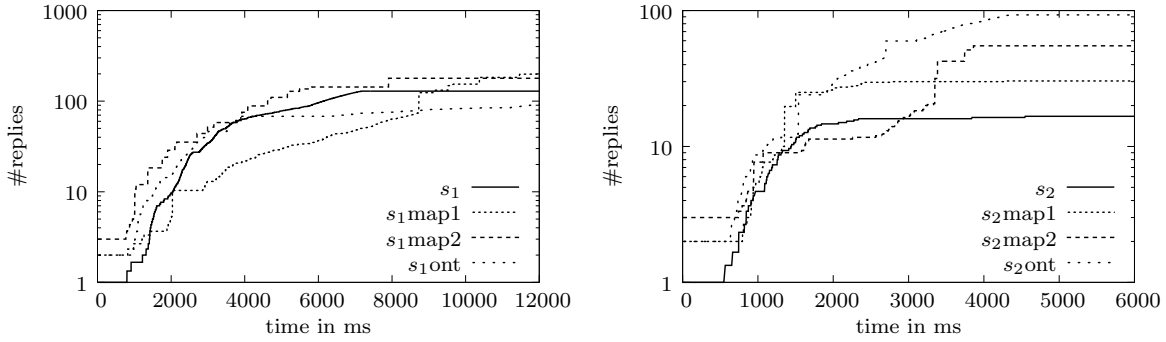


(a) With ω^{RQ} (b) With $\omega^{HASH,PAR}$

Figure 8.52: Aggregation query with the use of completeness estimation

Summarizing, we were able to show the accuracy of the proposed method for estimating query completeness. By introducing three different techniques, we are also able to determine guaranteed lower and upper bounds. The resulting curves show either a straight line or a lightning-like shape, getting the more escalating the higher the degree of parallelism (and data-dependence) gets. The curves produced by Seaweed [145] show a Z-shape instead. This is due to the fact that Seaweed aims at providing delay-aware querying while focusing on aggregate queries. Downtime of unavailable peers is predicted and results are gathered when this time passes. In contrast, UniStore follows a best-effort approach, which is also reflected in the dynamic and online character of its completeness estimation.

8.2.6 Integrating Semantics



(a) s_1 expanded on 70 nodes

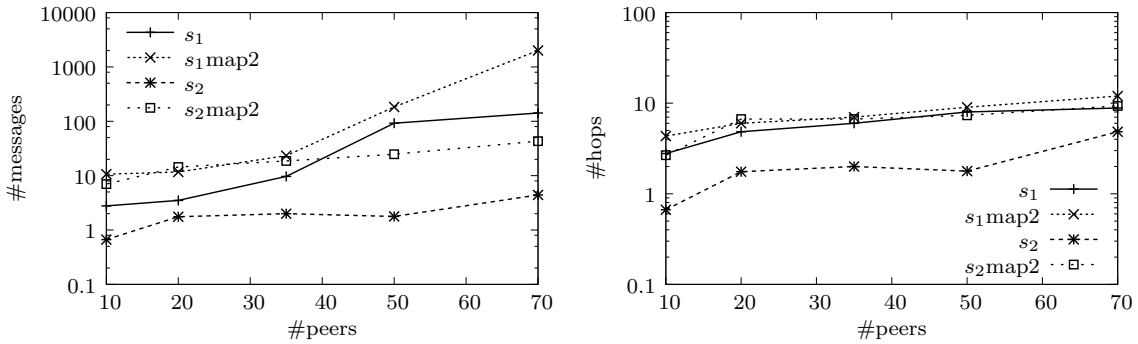
(b) s_2 expanded on 70 nodes

Figure 8.53: Query expansion

The objective of the last part of the evaluation is to provide a proof of concept of the approach for integrating semantic correspondences into query processing. For this, we show that we can achieve our main requirement, which is efficiency in query processing. We show that we first meet our main requirement, not to influence performance of an unexpanded query. Further, the following results document that query expansion is processed at a very low overhead and perfectly integrated into parallelized query processing.

We ran our tests on a network of up to 70 instances in our local environment. We used the geographical data from the DBPedia and Mondial data sets, enriched by small manually defined correspondences. These correspondences are stored in triples containing the `map:equiv` attribute, as introduced in Section 4.2.3. Further, we use small ontology data to express complex correspondences. These data also conform to the examples introduced in Section 4.2.3. From this set of about 30000 triples, we inserted about 7000 randomly chosen triples into the system. On these triples, we built two indexes, one on the concatenation of attribute names and values and another on the identifiers (OIDs). Thus, we handled about 14000 triples in the system. In the following, we present a selected subset of all results gathered. In each test, we ran a set of chosen queries, each initiated 5 times by randomly chosen peers. The two queries chosen for representation correspond to the semantic class s presented in Section 8.2.1. Results for other queries look similar.

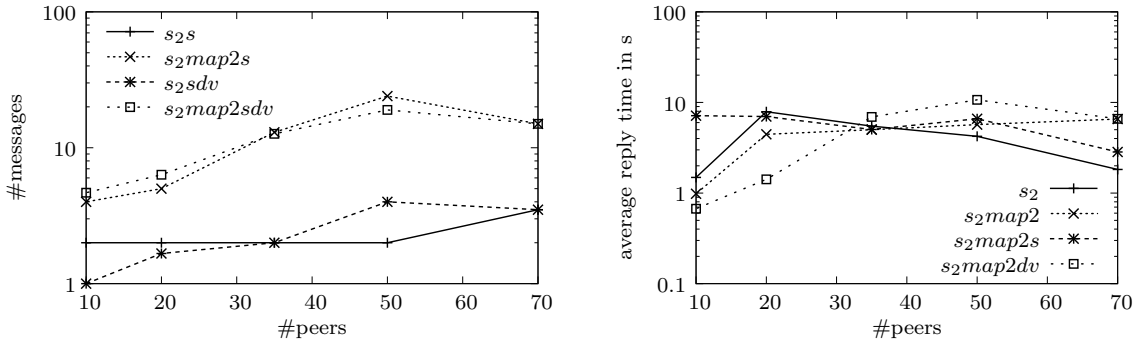
The first results show the performance of queries on the largest network size of 70 nodes. In Figure 8.53 we plot the number of replies received over time. We plot the number of replies rather than the result size, because this size depends too much on the query and data selectivity. The result size increases nearly in parallel to the number of replies, but showing more leaps. The number of replies (including empty replies) gives a better insight on the actual performance. First, the queries s_1 and s_2 were run separately. Afterwards, each query was expanded by one mapping (s_*map1) and two mappings (s_*map2). Further, we used the ontology query used in Section 5.1.2 (Figure 5.5, page 83) for expanding the original queries, represented by s_*ont. Queries were run using only parallel operators. Both plots clearly show that running an expanded query does not influence response time negatively – rather, in some cases the results of expanded queries arrive earlier. All in all, the plot for the number of replies shows a similar shape for all queries, all the same how much parallelism is involved or how complex the query is (e.g., s_*ont).



(a) Messages (b) Hops

Figure 8.54: Overhead

This is due to the parallel processing approach, which comes at the costs of messages. This is illustrated in Figure 8.54(a) (note the logarithmic scale). The number of messages increases almost exponentially with the network size for some of the queries. For others, the overhead is constant. This mainly depends on the number of intra-operator parallel processing stages that a query plan contains. Bandwidth consumption raises almost in the same scale like the number of messages. The though low response times are reflected by the number of hops shown in Figure 8.54(b).



(a) Varied parallelism (b) Varied data volume

Figure 8.55: Performance under different variations

There may be a point where such an amount of generated messages results in performance leaks. The more parallel operators in a post-order plan, the higher the number of messages. This can be avoided by increasing inter-operator parallelism (process more sub-trees in parallel) or by using alternative operators in specific situations (this decision is due to a sophisticated optimizer). We show this for s_2 in Figure 8.55(a). $s_2 * s$ are the same queries as before, but all materialize operations are run using a sequential operator, which decreases parallelism. The number of generated messages clearly drops. Finally, we show that the size of the inserted data has very low impact on the query performance: queries $s_2 * sdv$ are the same as $s_2 * s$, but were run using increasing data volume with increasing network size (each node inserted the same amount of triples, up to the size of 14000 handled triples). Response times for all queries

are very satisfactory, as Figure 8.55(b) shows. We plot the time averaged over all replies for some selected queries. The time for the last result increases linearly with the network size, as expected. The time for the first result behaves very similar like the average time plotted – which is backed by the plots in Figures 8.53(a) and 8.53(b). Most of all results and replies are received in the first seconds after issuing a query. With respect to the network size, response times fortunately even decrease.

All in all, the presented results only give an insight in actual performance of query processing. But, they absolutely justify the statement that the proposed approach fits well into the implemented query processing strategies. With this foundation, querying a heterogeneous, semantically enriched data storage emerges as simple and efficient as querying the Web itself, but also as powerful as querying database systems. This all comes with low effort and no expert user knowledge or intervention needed at all.

8.2.7 Summary

In the experiments, we were able to show that the proposed framework for query processing is a powerful and very promising approach for enabling Public Data Management. With the presented techniques, it is possible to fairly distribute the costs of query processing among all participants. This is due to solving some of the main challenges discussed in this work: a wide variety of sophisticated query constructs is efficiently supported. Data and load are fairly distributed. Further, this comes along with several important guarantees. On the one hand, this enables a meaningful and practical cost-based optimization, which is mandatory concerning the performance of operators depending on current network, data, and query characteristics. On the other hand this enables, meaningful querying even in widely distributed, dynamic, and unreliable environments. The experimental results for completeness estimation show that this is a very promising approach to provide a kind of guarantees even in systems that originally can provide nothing similar.

Concerning actual query processing, we showed what we expected before. Parallel strategies are faster, more reliable, but also more expensive than sequential ones. They are particularly suited for a high distribution of query load and offer themselves especially in unreliable environments. But, also sequential approaches can be advantageous in specific situations. They are suited to decrease the total load injected into the system. This is especially true for hash-based operators on small input data. Sequential processing strategies are particularly suited for reliable environments with fast network connections.

Alternative operators provide a valuable extension to the set of standard distributed processing approaches. We showed this for the q-gram operators. They do not only support similarity on arbitrary levels very well, they also allow for a range of other efficient operator implementations. If according queries are expected to be frequent, the costs for building the needed q-gram indexes should be worth. This is especially true as standard operators are not suited to support the ad-hoc handling of schema similarity, an extremely important facet of Public Data Management. Nevertheless, similarity on instance level is backed by other operators efficiently as well.

The exact costs and benefits of the operators and processing strategies depend on a wide range of factors. This involves the actual queries and the resulting combination of operators, network characteristics, data distribution, etc. We were able to show that the proposed cost model provides a satisfying and promising basis for handling these issues in a meaningful way. However, we also showed that there are still several unsolved issues in the context of cost estimations. The experimental results provide insights into further aspects in this context and outline directions for further research in this area.

All in all, the query processing framework solves a couple of the main challenges of large-scale distributed data management. Thus, it provides an important foundation for enabling database-oriented Public Data Management applications, in which the costs of the infrastructure can be shared among computers from the edge of the Internet. The important requirement of providing a set of basic guarantees concerning quality and efficiency of query processing is fulfilled as well.

Chapter 9

Conclusion and Outlook

In this work, we discussed a wide range of issues and challenges. We detailedly presented approaches and solutions for the main challenges. We finally summarize and conclude the work, its contributions, and the discussed and proposed concepts and approaches in Section 9.1. Nevertheless, there are still several open issues that mark the way for future research in this area. In Section 9.2, we briefly summarize the most important of them.

9.1 Summary

We discussed the wide range of challenges of enabling large-scale distributed data management with database-like features. We cited novel applications like Public Data Management scenarios as examples and driving factors for the increasing need for such infrastructures. We classified the challenges and identified those that specifically relate to processing structured queries that support sophisticated and specialized query constructs on structured data. From a classification of existing architectural approaches for widely distributed data systems we distinguished the P2P paradigm as the most promising one. Based on this and basic suggestions for universal and flexible data management, we focused on the issues of providing a query processing framework suited to enable meaningful database-like query capabilities. Although we focused on Public Data Management as an example type of application, the discussions and results provide valuable insights and cognitions for large-scale data management in general.

We indicated ways and developed approaches for enabling totally decentralized data managing and querying. The proposed framework with its heart, the physical query algebra based on vertical triple indexes, represents a more than promising foundation for supporting meaningful and efficient distributed query processing. We identified the main novel processing paradigms and successfully combined them with the cognitions from traditional database query processing. A decentralized optimization taking special aspects of distributed systems into account was identified as a mandatory requirement in this context. The discussed cost model shows that techniques known from database systems can be adopted for this purpose as well. The problems and challenges of enabling accurate cost estimations are even increased in the decentralized distributed setup. However, integrating popular approaches for estimating selectivities of filter expressions and q-grams promise to make the cost model adequate for utilization. We showed that even in its basic form the model contributes significantly to a meaningful and feasible query planning.

In analogy to the suggested universal and flexible data model, the query processing strategies and the query framework in general are generic, universal, and flexible as well. The evaluation showed that the proposed framework is suited to be applied in widely distributed,

decentralized, and even unreliable and dynamic environments. Even predictability and guarantees are enabled through an efficient approach for estimating query completeness. Existing distributed processing approaches have been adopted and extended by novel techniques successfully. The evaluation showed the known and expected trade-offs between parallel and sequential approaches. Further, some new insights into the possibilities of distributed processing have been highlighted. For instance, the hash-based and q-gram-based operators, which enable efficient similarity queries on different levels and complex database-like queries. The benefits of strictly stateless processing techniques have been identified and implemented.

The main challenge solved in this work is that we provide a query framework for efficient processing that backs the leveraged query expressiveness of the underlying query language. This enables a universal and flexible storage system that comes into existence only through collaboration of its participants. Handling the implicit heterogeneities is provided implicitly as well as explicitly by the proposed approach. We were able to show that traditional database techniques that proved their feasibility can also be implemented in widely distributed systems, with only specific relaxations rather than dropping them at all. Summarizing, we provide and implemented a valuable basis for a truly scalable universal storage system. This enables meaningful Public Data Management through distributing data, load, and thus, costs among all participants from the edge of the Internet. By this, the chances and options that collaboration provides are utilized to a novel degree. From our point of view, there is no other approach that promises a similar scalability while supporting such a fair distribution and still providing a basic subset of required guarantees and qualities.

A very crucial issue are guarantees in the proposed system architecture. The approach for estimating query completeness shows that guarantees can be provided, even if we adhere to a totally decentralized and autonomous query processing. This reveals that the aimed system approach has not to neglect guarantees, rather they have to be relaxed. The idea of probabilistic guarantees emerged as a promising and in cases inevitably alternative.

9.2 Open Issues and Future Directions

First, the suggestions made as a basis for discussing distributed query processing require deeper investigation. On the distribution layer, DHTs have been researched thoroughly. However, the P2P paradigm and general issues and benefits of decentralized distributed systems are still in the focus of current research. Recent developments like the emerging cloud services will benefit from according activities as well as provide valuable cognitions for driving them. A combination of the approach proposed in this work and the cloud computing idea would result in a kind of cloud database service. Further research on the specific challenges and chances of this storage service approach is a very interesting and prevailing direction.

To enable such a service, it is mandatory to discuss the provision of guarantees. According to the CAP theorem, trade-offs between the possible guarantees and other design goals like availability and efficiency have to be identified. Current cloud services act like a “decentralized central instance” to support, for instance, consistency. This still limits them in their scalability and thus, it narrows the range of benefiting applications. A very interesting approach is to analyze Quality-of-Service (QoS) guarantees in the context of extremely distributed and decentralized systems. Guarantees like the proposed estimated query completeness are only a first step towards QoS-enabled applications. This further spans to predict answer times, availability, freshness, etc. In this context, the proposed approach requires some further investigation concerning efficient and accurate estimations on data level. Further, different distribution models for determining probabilistic guarantees have to be evaluated. However, it is more important

to discuss further approaches for predictability and guarantees, for instance in the context of availability, freshness, and consistency in the presence of updates. Supporting efficient and consistent update functionality is a main requirement for enabling full-fledged database applications.

The suggested vertical data model and the accompanying query language plus algebra represent very interesting and promising approaches. Analyzing and evaluating their advantages and disadvantages is another main open issues. This should be accompanied by exploring promising possibilities provided by the data model, such as explicitly utilizing redundancies in indexing (a kind of index replication), dynamic and adaptive indexing techniques, mappings of a wider range of data models, further leveraged expressiveness of queries and query constructs, etc.

Semantic Web applications are in the focus of a lot of recent activities. Cognitions about the managing and handling of heterogeneities and according complex correspondence graphs are very important for enabling Public Data Management. We provided a valuable basis for integrating semantic correspondences and integration tasks into query processing. However, the proposed semantic layer needs further investigation, extension, and improvement.

We did not discuss issues of trust, security, reputation, and privacy in detail. Throughout the work, we briefly pointed to works dealing with these issues. For enabling meaningful Public Data Management that is accepted and adopted by a wide range of users, these issues demand for intensive research.

Several minor issues of query processing need further investigation as well. This includes deeper activities concerning cost-based optimizations. The cost model presented in this work provides a valuable basis for further discussions in this direction. The evaluation proved that still a lot of aspects and issues that influence actual costs of query processing have to be investigated with increased effort. Analyzing and improving the process of logical optimization should come along with research in the area of query language and algebra. But, also the handling of query plans during query runtime needs further efforts. This comprises the utilization and provision of variables bindings, the handling of shared expressions and shared intermediate results, etc. Also, the evaluation indicated that aspects like availability and dynamics can influence query performance significantly. Consequently, they should be integrated into the cost model. This requires deeper investigation of the underlying relations and mutual effects. Further, quality measures especially relevant in loosely-couple distributed systems, such as the achievable robustness, have to be integrated as cost factors into the model. Sophisticated plan processing strategies require more development as well. Adaptive processing approaches have been discussed, but not fully analyzed nor evaluated. A detailed comparison and analysis of approaches like Eddies promise to provide very interesting and valuable insights and cognitions. The application of aggregation trees and utilization of multiple synchronization peers are further open aspects. Special operator concepts like skyline querying could benefit from a detailed comparison of alternative approaches. In the context of skyline queries this includes the comparison of different approaches for narrowing the processed search space.

The concepts and techniques of this work could be proven to be applicable and efficient in the evaluation. Also the correctness of the cost model could be indicated. Nevertheless, a deeper investigation and extended experimentation are limited in scale due to the available testing platforms. Finding a way for running real large-scale experiments behind the size of Planet-Lab is a prerequisite for that. Preferably, a corresponding test-bed should be more robust and suited for data-oriented applications. However, a perfect test would be to use the implemented prototype to develop an according full-fledged client that is used in a concrete Public Data Management application. With the help of millions of Internet users, a thorough analysis would be enabled. The emerging popularity of social and semantic applications in the

Web will help to identify according concrete scenarios. Moreover, the discussed concepts should be analyzed concerning their suitability behind the borders of Public Data Management.

Appendix A

Basic VQL Syntax

In the following, we define the basic syntax of VQL as it is applied in this work using the well-known Extended Backus-Naur Form (EBNF). For convenience and readability, we omit any white space characters that should be placed between symbols (terminal and non-terminal), because we regard this as intuitive. Further, concatenation of symbols is indicated by a white space rather than by ",":

```
VQL-QUERY ::= "select" ["distinct"] VAR-LIST "where" "{"
              WHERECLAUSE "}";
```

Variables are identified by a leading question mark and are used in the triple patterns that are contained in `TRIP-PATT-LIST` and `FILT-LIST`. `VAR-LIST` is a list of variables that shall be projected from the final result set:

```
VAR-LIST ::= VAR[PROJ] | VAR[PROJ] "," VAR-LIST
PROJ := "*" | "*.*" | ".*";
VAR ::= "?[a-zA-Z][a-zA-Z0-9]*";
```

The definition for `VAR` combines EBNF with regular expressions, which is for convenience again. The special projection `?var*.*` projects all attributes belonging to the same OID – only variables referencing OIDs are allowed here. Optional conditions that the materialized triples have to adhere to are provided by corresponding triple patterns in the `WHERE` clause. If the keyword `distinct` is provided in the `SELECT` clause, duplicate result rows will be filtered out.

The triple patterns from `TRIP-PATT-LIST` declare what data are actually contained in the result set:

```
WHERECLAUSE ::= PART | PART "union" WHERECLAUSE;
PART ::= TRIP-PATT-LIST [FILT-LIST];
TRIP-PATT-LIST ::= TRIP-PATT | TRIP-PATT "." TRIP-PATT-LIST;
```

Each `TRIP-PATT` defines a condition that all contained triples have to match:

```
TRIP-PATT ::= E E E;
E ::= VAR | CONST;
CONST ::= ["-"] "[[:digit:]]+" | "'[[:print:]]*'";
```

Constants may be numerical values or string constants. String constants have to be embraced by '. Here, we use regular expressions in combination with POSIX character classes to shorten the defining rule. The definition allows only integer values as numerical constants, but floating

point constants can be used in VQL as well. The `FILT-LIST` contains additional expressions that are used to filter the input data:

```
FILT-LIST ::= "filter" EXPR | "filter" EXPR FILT-LIST;
EXPR ::= VAR OP CONST | VAR OP VAR |
         FUNC"("VAR","CONST")" OP (CONST | BOOL) |
         FUNC"("VAR","VAR")" OP (CONST | BOOL);
OP ::= "=" | "<" | ">" | "<=" | ">=" | "!=";
BOOL ::= "true" | "false";
FUNC ::= "edist" | ... ;
```

Note that boolean values can only be compared by '=' and '!=', which is not guaranteed by the simplified syntax from above. In contrast to SPARQL, all filter expressions are listed after all triple patterns. They cannot be mixed up. This is just for implementation reasons [190] and makes no restriction. There are a couple of functions already integrated and they can be extended at will. Here we only list the *edist* function, which computes the edit distance (or Levenshtein distance [126]) between two strings.

The syntax for further constructs, such as `ORDER BY`, `SKYLINE`, `LIMIT`, etc., is omitted. More details about the VQL syntax (in an earlier version) can be found in [190].

Bibliography

[1] K. Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *CoopIS'01*, pages 179–194, 2001.

[2] K. Aberer. Scalable Data Access in P2P Systems Using Unbalanced Search Trees. In *Workshop on Distributed Data and Structures (WDAS'02)*, 2002.

[3] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. *SIGMOD Record*, 32(3):29–33, 2003.

[4] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, R. Schmidt, and J. Wu. Advanced peer-to-peer networking: The P-Grid System and its Applications. *PIKJ, Spec. Issue on P2P Systems*, 26:86–89, 2003.

[5] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth. Start making sense: The chatty web approach for global semantic agreements. *Journal of Web Semantics*, 1(1):89–114, 2003.

[6] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, and T. Van Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In *ISWC'04*, pages 107–121, 2004.

[7] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt. Indexing data-oriented overlay networks. In *VLDB'05*, pages 685–696, 2005.

[8] K. Aberer and M. Hauswirth. P2P Information Systems: Concepts and Models, State of the Art, and Future Systems. In *ICDE'02*, page 187, 2002. Advanced Technology Seminar, slides available at http://www.manfredhauswirth.org/research/papers/ICDE2002-Tutorial.pdf.

[9] K. Aberer, M. Hauswirth, and M. Punceva. Self-organized construction of distributed access structures: A comparative evaluation of P-Grid and Freenet. In *5th Workshop on Distributed Data and Structures (WDAS'2003)*, 2003.

[10] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *Mobile Data Management (MDM'07)*, 2007.

[11] K. Aberer and M. Punceva. Efficient Search in Structured Peer-to-Peer Systems: Binary v.s. k-ary Unbalanced Tree Structures. In *Int. Workshop On Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'03)*, 2003.

[12] E. Adar and B.A. Huberman. Free Riding on Gnutella. *First Monday*, 5(10), 2000.

[13] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[14] St. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36:335–371, 2004.

[15] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The Hyperion Project: From Data Integration to Data Coordination. *SIGMOD Record*, 32(3):53 – 58, 2003.

[16] J. Aspnes, J. Kirsch, and A. Krishnamurthy. Load balancing and locality in range-queriable data structures. In *PODC'04*, pages 115–124, 2004.

[17] J. Aspnes and G. Shah. Skip graphs. In *Symposium on Discrete Algorithms*, pages 384–393, 2003.

[18] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD '00*, pages 261–272, 2000.

[19] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, 2003.

[20] W. Balke, U. Güntzer, and J. Xin Zheng. Efficient Distributed Skylining for Web Information Systems. In *EDBT'04*, pages 256–273, 2004.

[21] W. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive distributed top k retrieval in peer-to-peer networks. In *ICDE'05*, pages 174–185, 2005.

[22] F. Banaei-Kashani and C. Shahabi. SWAM: a family of access methods for similarity-search in peer-to-peer data networks. In *CIKM'04*, pages 304–313, 2004.

[23] St. Battle. Gloze: XML to RDF and back again. In *First Jena User Conference*, 2006.

[24] D. Bauer, P. Hurley, R. Pletka, and M. Waldvogel. Bringing Efficient Advanced Queries to Distributed Hash Tables. In *Local Computer Networks (LCN'04)*, pages 6–14, 2004.

[25] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *WWW'05*, pages 651–660, 2005.

[26] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. *Journal of Computer and System Sciences*, 73(3):245–264, 2007.

[27] Ph. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing: A vision. In *Fifth Int. Workshop on the Web and Databases (WebDB'02)*, pages 89–94, 2002.

[28] R. Bhagwan, D. Moore, St. Savage, and G. M. Voelker. Replication Strategies for Highly Available Peer-to-Peer Storage. In *Future directions in Distributed Computing*, pages 153–158, 2002.

[29] A.R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. *SIGCOMM Computer Communication Review*, 34(4):353–366, 2004.

[30] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. *XQuery 1.0: An XML Query Language (W3C Working Draft 16 August 2002)*. W3C, 2002. http://www.w3.org/TR/xquery/.

[31] P. Boncz and C. Treijtel. AmbientDB: Relational Query Processing in a P2P Network. In *Workshop On Databases, Information Systems and P2P Computing, (DBISP2P'03)*, pages 153–168, 2003.

[32] A. Bonifati, P. K. Chrysanthis, A. M. Ouksel, and K. Sattler. Distributed databases and peer-to-peer databases: past and present. *SIGMOD Record*, 37(1):5–11, 2008.

[33] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE'01*, pages 421–432, 2001.

[34] A. Bouguettaya, B. Benatallah, and A. Elmagarmid. *An overview of multidatabase systems: past and present*, pages 1–32. Morgan Kaufmann Publishers Inc., 1999.

[35] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD '08*, pages 251–264, 2008.

[36] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *VLDB Journal*, 10(1):48–71, 2001.

[37] J. Brevik, D. Nurmi, and R. Wolski. Automatic Methods for Predicting Machine Availability in Desktop Grid and Peer-to-Peer Systems. In *Int. Symposium on Cluster Computing and the Grid*, pages 190–199, 2004.

[38] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC'00*, page 7, 2000. Key note talk, slides available at http://www.cs.berkeley.edu/∼brewer/cs262b-2004/PODC-keynote.pdf.

[39] I. Brunkhorst, H. Dhraief, A. Kemper, W. Nejdl, and Ch. Wiesner. Distributed Queries and Query Optimization in Schema-Based P2P Systems. In *International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'03)*, 2003.

[40] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *ACM TODS*, 27(2):153–187, 2002.

[41] M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW'04*, pages 650–657, 2004.

[42] M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In *Int. Workshop on Grid Computing (GRID'03)*, page 184, 2003.

[43] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC'04*, pages 206–215, 2004.

[44] J. Cappos and J. H. Hartman. San fermín: aggregating large data sets using a binomial swap forest. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, pages 147–160, 2008.

[45] M. Castro, M. Costa, and A. I. T. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *DSN'04*, pages 9–18, 2004.

[46] St. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *SIGMOD'82*, pages 128–136, 1982.

[47] D. D. Chamberlin and R. F. Boyce. SEQUEL: A structured English query language. In *Workshop on Data description, access and control (FIDET'74)*, pages 249–264, 1974.

[48] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):1–26, 2008.

[49] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating Progress of Long Running SQL Queries. In *SIGMOD'04*, pages 803–814, 2004.

[50] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[51] Ciao.com homepage. http://www.ciao.com (last visited 2009/02/21).

[52] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0 (W3C Recommendation 16 November 1999)*. W3C, 1999. http://www.w3.org/TR/xpath.

[53] I. Clarke, O. Sandberg, B. Wiley, and Th. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[54] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 26(1):64–69, 1983.

[55] W. W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems*, 18(3):288–321, 2000.

[56] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Int. Conf. on Distributed Computing Systems*, pages 23–32, July 2002.

[57] P. Cudré-Mauroux, K. Aberer, and A. Feher. Probabilistic message passing in peer data management systems. In *ICDE'06*, page 41, 2006.

[58] P. Cudré-Mauroux, S. Agarwal, and K. Aberer. GridVine: an Infrastructure for Peer Information Management. *IEEE Internet Computing*, 11(5):864–875, 2007.

[59] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *ICDCS'03*, page 76, 2003.

[60] A. Datta, M. Hauswirth, R. Schmidt, R. John, and K. Aberer. Range queries in trie-structured overlays. In *P2P'05*, pages 57–66, 2005.

[61] IBM DB2 homepage. http://www-01.ibm.com/software/data/db2/9/ (last visited 2008/11/8).

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

[63] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Operation Systems Review*, 41(6):205–220, 2007.

[64] St. Decker and M. Hauswirth. Enabling Networked Knowledge. Technical Report 2008-06-25, Digital Enterprise Research Institute (DERI), June 2008.

[65] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic Data Publication over the Internet. In *IFIP Working Conference in Database Security*, pages 102–112, 2000.

[66] Ebay homepage. http://www.ebay.com (last visited 2008/11/12).

[67] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS'01*, pages 102–113, 2001.

[68] Friend of a Friend homepage. http://www.foaf-project.org (last visited 2009/02/21).

[69] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure (2nd ed.)*. Elsevier, 2004.

[70] Freenet Project. http://FreenetProject.org (last visited 2009/02/21).

[71] L. Galanis, Y. Wang, Sh. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB'03*, pages 874–885, 2003.

[72] S. Ghemawat, H. Gobioff, and Sh.-T. Leung. The Google file system. *SIGOPS Operation Systems Review*, 37(5):29–43, 2003.

[73] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[74] Gnutella homepage. http://rfc-gnutella.sourceforge.net (last visited 2008/11/6).

[75] Google homepage. http://www.google.com (last visited 2008/11/6).

[76] A. Gounaris, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou. Adaptive Query Processing: A Survey. In *BNCOD'02*, pages 11–25, 2002.

[77] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *VLDB'01*, pages 491–500, 2001.

[78] S. D. Gribble, A. Y. Halevy, Z. G. Ives, M. Rodrig, and D. Suciu. What Can Database Do for Peer-to-Peer? In *WebDB'01*, pages 31–36, 2001.

[79] T. Groza, S. Handschuh, K. Moeller, G. Grimnes, L. Sauermann, E. Minack, C. Mesnage, M. Jazayeri, G. Reif, and R. Gudjonsdottir. The NEPOMUK Project - On the way to the Social Semantic Desktop. In *Int. Conferences on new Media technology (I-MEDIA'07) and Semantic Systems (I-SEMANTICS'07)*, pages 201–210, 2007.

[80] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In *CIDR'03*, pages 141–151, 2003.

[81] Darwen H and C. J. Date. The third manifesto. *SIGMOD Record*, 24(1):39–49, 1995.

[82] Hadoop homepage. http://hadoop.apache.org (last visited 2008/11/7).

[83] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[84] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *WWW'03*, pages 556–567, 2003.

[85] N. J. A. Harvey, M. B. Jones, St. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USENIX Symposium on Internet Technologies and Systems (USITS'03)*, 2003.

[86] HBase homepage. http://hadoop.apache.org/hbase (last visited 2008/11/7).

[87] F. Heine. Scalable p2p based RDF querying. In *Int. Conference on Scalable Information Systems (InfoScale'06)*, pages 17–22, 2006.

[88] F. Heine, M. Hovestadt, and O. Kao. Processing complex RDF queries over P2P networks. In *Workshop on Information retrieval in peer-to-peer networks (P2PIR'05)*, pages 41–48, 2005.

[89] J. M. Hellerstein. Toward network data independence. *SIGMOD Record*, 32(3):34–40, 2003.

[90] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD'97*, pages 171–182, 1997.

[91] K. Hose. *Processing Rank-Aware Queries in Schema-Based P2P Systems.* PhD thesis, Ilmenau University of Technology, 2009.

[92] 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.

[93] K. Hose, M. Karnstedt, K. Sattler, and D. Zinn. Processing Top-N Queries in P2P-based Web Integration Systems with Probabilistic Guarantees. In *WebDB'05*, pages 109–114, June 2005.

[94] K. Hose, C. Lemke, and K. Sattler. Processing Relaxed Skylines in PDMS Using Distributed Data Summaries. In *CIKM'06*, pages 425–434, 2006.

[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, Ch. S. Jensen, H. Lu, and B. C. Ooi. Skyline Queries Against Mobile Lightweight Devices in MANETs. In *ICDE'06*, page 66, 2006.

[98] R. Huebsch, B. Chun, and J. M. Hellerstein. Pier on PlanetLab: Initial experience and open problems. Technical report, University of California, Berkeley, 2003.

[99] R. Huebsch, B. Chun, J. M. Hellerstein, B. Th. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. In *CIDR'05*, pages 28–43, 2005.

[100] R. Huebsch, J. M. Hellerstein, N. Lanham, B. Thau Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB'03*, pages 321–332, 2003.

[101] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD'91*, pages 268–277, 1991.

[102] H. V. Jagadish, B. Ch. Ooi, and Q. H. Vu. BATON: a balanced tree structure for peer-to-peer networks. In *VLDB'05*, pages 661–672, 2005.

[103] T. Jim and D. Suciu. Dynamically Distributed Query Evaluation. In *PODS'01*, pages 28–39, 2001.

[104] M. Karnstedt, J. Müller, and K. Sattler. Cost-Aware Skyline Queries in Structured Overlays. In *ICDE Workshop on Ranking in Databases (DBRank'07)*, pages 285–288, 2007.

[105] M. Karnstedt, K. Sattler, M. Haß, M. Hauswirth, B. Sapkota, and R. Schmidt. Approximating Query Completeness by Predicting the Number of Answers in DHT-based Web Applications. In *Int. Workshop on Web Information and Data Management (WIDM'08)*, pages 71–78, 2008.

[106] M. Karnstedt, K. Sattler, M. Hauswirth, and R. Schmidt. Cost-Aware Processing of Similarity Queries in Structured Overlays. In *P2P'06*, pages 81–89, 2006.

[107] M. Karnstedt, K. Sattler, M. Hauswirth, and R. Schmidt. Similarity Queries on Structured Data in Structured Overlays. In *Int. Workshop on Networking Meets Databases (NetDB'06)*, pages 32–37, 2006.

[108] M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John. UniStore: Querying a DHT-based Universal Storage. In *ICDE'07 Demonstrations Program*, pages 1503–1504, 2007.

[109] M. Karnstedt, K. Sattler, and R. Schmidt. Completeness Estimation of Range Queries in Structured Overlays. In *P2P'07*, pages 71–78, 2007.

[110] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: a declarative query language for RDF. In *WWW'02*, pages 592–603, 2002.

[111] KaZaA homepage. http://www.kazaa.com (last visited 2008/11/6).

[112] A. Kemper and C. Wiesner. Hyperqueries: Dynamic Distributed Query Processing on the Internet. In *VLDB'01*, pages 551–560, 2001.

[113] R. Khoussainov and N. Kushmerick. Automated index management for distributed web search. In *CIKM'03*, pages 386–393, 2003.

[114] D. Klan, K. Sattler, K.Hose, and M. Karnstedt. Decentralized Managing of Replication Objects in Massively Distributed Systems,. In *EDBT Workshop on Data Management in Peer-to-peer systems (DaMaP'08)*, pages 19–26, 2008.

[115] J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *ACM STOC*, pages 163–170, 2000.

[116] K. Kochut and M. Janik. SPARQLeR: Extended Sparql for Semantic Association Discovery. In *ESWC'07*, volume 4519, pages 145–159, 2007.

[117] G. Kokkinidis, E. Sidirourgos, and V. Christophides. Query Processing in RDF/S-Based P2P Database Systems. In S. Staab and H. Stuckenschmidt, editors, *Semantic Web and Peer-to-Peer*, chapter 4, pages 59–81. Springer, Berlin Heidelberg, 2006.

[118] G. Koloniari and E. Pitoura. Peer-to-Peer Management of XML Data: Issues and Research Challenges. *SIGMOD Record*, 34(2):6–17, 2005.

[119] M. Kornfilt and M. Hauswirth. Efficient processing of rare queries in Gnutella using a hybrid infrastructure. In *Int. Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'07)*, 2007.

[120] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[121] D. Kossmann. A Robust Spanning Tree Topology for Data Collection and Dissemination in Distributed Environments. *IEEE Transactions on Parallel and Distributed Systems*, 18(5):608–620, 2007.

[122] 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.

[123] J. Kubiatowicz. Extracting guarantees from chaos. *Communications of the ACM*, 46(2):33–38, 2003.

[124] J. Kubiatowicz, D. Bindel, Y. Chen, St. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, Ch. Wells, and B. Zhao. OceanStore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, 2000.

[125] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser. StreamGlobe: Processing and Sharing Data Streams in Grid-Based P2P Infrastructures. In *VLDB'05*, pages 1259–1262, 2005.

[126] V. Levenshtein. Binary codes of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[127] A. Y. Levy. Obtaining Complete Answers from Incomplete Databases. In *VLDB'96*, pages 402–412, 1996.

[128] H. Li, Q. Tan, and W.-C. Lee. Efficient progressive processing of skyline queries in peer-to-peer systems. In *1st Int. conference on Scalable information systems (InfoScale'06)*, page 26, 2006.

[129] X. Li and J. Wu. Searching Techniques in Peer-to-Peer Networks. In J. Wu, editor, *Handbook of Theoretical and Algorithmic Aspects of Ad Hoc, Sensor, and Peer-to-Peer Networks*, chapter 3. Auerbach Publications, Boca Raton, FLorida, USA, 2006.

[130] Ch. Y. Liau, W. S. Ng, Y. Shu, K.-L. Tan, and St. Bressan. Efficient range queries and fast lookup services for scalable p2p networks. In *Int. Workshop On Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'04)*, pages 93–106, 2004.

[131] B. Th. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P file-sharing with an internet-scale query processor. In *VLDB '04*, pages 432–443, 2004.

[132] K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, 7:72–93, 2005.

[133] J. Madhavan, Sh. R. Jeffery, Sh. Cohen, X. L. Dong, D. K., C. Yu, and A. Halevy. Web-scale data integration: You can only afford to pay as you go. In *CIDR'07*, pages 342–350, 2007.

[134] A. Magkanaraki, V. Tannen, and V. Christophides. Viewing the Semantic Web Through RVL Lenses. In *ISWC'03*, pages 98–112, 2003.

[135] D. Maier, J. D. Ullman, and M. Y. Vardi. On the Foundations of the Universal Relation Model. *ACM TODS*, 9(2):283–308, 1984.

[136] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM TODS*, 29(2):319–362, 2004.

[137] L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A.J. Ganesh. Peer Counting and Sampling in Overlay Networks: Random Walk Methods. *PODC'06*, pages 123–132, 2006.

[138] K. Mathaj. Semantische Interoperabilität in UniStore, 2008. Diploma Thesis at TU Ilmenau (available in German only).

[139] S. Melnik. Bridging the Gap between RDF and XML, 1999. http://www-db.stanford.edu/∼melnik/rdf/fusion.html (last visited 2009/01/24).

[140] L. Miller, A. Seaborne, and A. Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *ISWC'02*, pages 423–435, 2002.

[141] A. Motro. Completeness Information and Its Application to Query Processing. In *VLDB'86*, pages 170–178, 1986.

[142] J. Müller. Berechnung von Skylines in strukturierten Overlaynetzwerken, 2007. Diploma Thesis at TU Ilmenau (available in German only).

[143] Myspace homepage. http://www.myspace.com (last visited 2009/02/21).

[144] Napster homepage. http://www.napster.com (last visited 2008/11/6).

[145] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with seaweed. In *VLDB'06*, pages 727–738, 2006.

[146] F. Naumann, J.-Ch. Freytag, and U. Leser. Completeness of integrated information sources. *Information Systems*, 29(7):583–615, 2004.

[147] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[148] G. Navarro and R. A. Baeza-Yates. A Practical q -Gram Index for Text Retrieval Allowing Errors. *CLEI Electron. Journal*, 1(2), 1998.

[149] Sh. Navathe, St. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM TODS*, 9(4):680–710, 1984.

[150] Y. Ndiaye, A. Wane Dine, W. Litwin, and T. Risch. Scalable Distributed Data Structures for High-Performance Databases. In *Int. Workshop on Distributed Data Structures (WDAS'00)*, 2000.

[151] 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 *WWW'02*, 2002.

[152] W. S. Ng, B. Ch. Ooi, and K.-L. Tan. BestPeer: A Self-Configurable Peer-to-Peer System. In *ICDE '02*, page 272, 2002.

[153] W. S. Ng, B. Ch. Ooi, and K.-L. Tan. PeerDB: A P2P-based System for Distributed Data Sharing. In *ICDE '03*, pages 633–644, 2003.

[154] Ch. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD'08*, pages 1099–1110, 2008.

[155] B. Ch. Ooi, Y. Shu, and K. L. Tan. DB-enabled peers for managing distributed data. In *Fifth Asia-Pacific Web Conference (APWeb'03)*, pages 10–21, 2003.

[156] B. Ch. Ooi, K.-L. Tan, A. Zhou, Ch. H. Goh, Y. Li, Ch. Y. Liau, B. Ling, W. S. Ng, Y. Shu, X. Wang, and M. Zhang. PeerDB: peering into personal databases. In *SIGMOD'03*, pages 659–659, 2003.

[157] B. Ch. Ooi, B. Yu, and G. Li. One table stores all: Enabling painless free-and-easy data publishing and sharing. In *CIDR'07*, pages 142–153, 2007.

[158] Oracle Database homepage, 2008. http://www.oracle.com/database/index.html (last visited 2008/11/8).

[159] M. Tamer Öszu and P. Valduriez. *Principles of distributed database systems.* Prentice Hall, Inc., 1991.

[160] OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/ (last visited 2009/02/21).

[161] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD'03*, pages 467–478, 2003.

[162] V. Papadimos and D. Maier. Distributed queries without distributed state. In *Fifth Int. Workshop on the Web and Databases (WebDB'02)*, pages 95–100, 2002.

[163] V. Papadimos and D. Maier. Mutant Query Plans. *Information and Software Technology*, 44(4):197–206, April 2002.

[164] Yahoo PNUTS homepage. http://research.yahoo.com/node/212 (last visited 2008/11/7).

[165] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, 2006. W3C Candidate Recommendation.

[166] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *ESWC'08*, pages 524–538, 2008.

[167] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief Announcement: Prefix Hash Tree. In *PODC'04*, page 368, 2004.

[168] M. Raskino, J. Fenn, and A. Linden. Extracting Value From the Massively Connected World of 2015. Technical Report G00125949, Gartner research, April 2005.

[169] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM'01*, 2001.

[170] S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Range Queries over DHTs. Technical Report IRB-TR-03-009, Intel Research Berkeley, Jun. 2003.

[171] RDF at W3C homepage. http://www.w3.org/RDF (last visited 2009/02/21).

[172] RDF-Schema at W3C homepage. http://www.w3.org/TR/rdf-schema (last visited 2009/02/21).

[173] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Technical Conference (ATEC'04)*, pages 10–10, 2004.

[174] M. Richtarsky. CouPe: A Query Processor for UniStore, 2007. Diploma Thesis at TU Ilmenau.

[175] J. Risson and T. Moors. Survey of Research towards Robust Peer-to-Peer Networks: Search Methods. Technical Report UNSW-EE-P2P-1-1, University of New South Wales, Sydney, Australia, Sep. 2004.

[176] J. Ritter. Why Gnutella Can't Scale. No, Really, 2001. http://www.darkridge.com/ jpr5/doc/gnutella.html (last visited 2008/11/06).

[177] P. Rösch, K. Sattler, C. von der Weth, and E. Buchmann. Best Effort Query Processing in DHT-based P2P Systems. In *ICDE'05 Workshops*, page 1186, 2005.

[178] J. W. Ross and G. Westerman. Preparing for utility computing: The role of IT architecture and relationship management. *IBM Systems Journal*, 43(1):5–19, 2004.

[179] A. Roth, F. Naumann, T. Hübner, and M. Schweigert. System P: Query Answering in PDMS under Limited Resources. In *WWW Workshop in Information Integration on the Web (IIWeb'06)*, 2006.

[180] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[181] Amazon S3 homepage. http://aws.amazon.com/s3 (last visited 2008/11/7).

[182] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A Peer-to-peer Framework for Caching Range Queries. In *ICDE'04*, page 165, 2004.

[183] A. Sahuguet, B. Pierce, and V. Tannen. Chaining, referral, subscription, leasing: New mechanisms in distributed query optimization, 2000. Online at: http://db.cis.upenn.edu/DL/new_mechanisms.pdf (last visited 2009/02/21).

[184] K. Sattler, I. Geist, and E. Schallehn. Concept-based Querying in Mediator Systems. *The VLDB Journal*, 14(1):97–111, 2005.

[185] K. Sattler, P. Rösch, E. Buchmann, and K. Böhm. A Physical Query Algebra for DHT-based P2P Systems. In *Workshop on Distributed Data and Structures (WDAS'04)*, 2004.

[186] Southern California Earthquake Center homepage. http://www.scec.org (last visited 2009/02/21).

[187] E. Schallehn, I. Geist, and K. Sattler. Supporting Similarity Operations based on Approximate String Matching on the Web. In *CoopIS'04*, pages 227–244, 2004.

[188] R. Schmidt. *Sharing of Probabilistically Correlated Data in Peer-to-Peer Networks.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), 2008.

[189] T. Scholl, B. Bauer, B. Gufler, R. Kuntschke, D. Weber, A. Reiser, and A. Kemper. HiSbase: histogram-based P2P main memory data management. In *VLDB'07*, pages 1394–1397, 2007.

[190] St. Schwalm. Anfragesystem für vertikal organisierte Universalrelationen in P2P-Systemen, 2006. Diploma Thesis at TU Ilmenau (available in German only).

[191] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Survey*, 22(3):183–236, 1990.

[192] W. Siberski, J. Pan, and U. Thaden. Querying the Semantic Web with Preferences. In *ISWC'06*, pages 612–624, 2006.

[193] Amazon SimpleDB homepage. http://aws.amazon.com/simpledb (last visited 2008/11/7).

[194] A. Singhal. Modern Information Retrieval: A Brief Overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4):35–42, 2001.

[195] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient Processing of XPath Queries with Structured Overlay Networks. In *OTM Conferences (2)*, pages 1243–1260, 2005.

[196] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01*, pages 149–160, 2001.

[197] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB'05*, pages 553–564, 2005.

[198] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed RDF repositories. In *WWW'04*, pages 631–639, 2004.

[199] T. Takaoka. Approximate Pattern Matching with Samples. In *Int. Symposium on Algorithms and Computation (ISAAC'94)*, pages 234–242, 1994.

[200] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, Inc., 2002.

[201] Ch. Tang, Z. Xu, and M. Mahalingam. Peersearch: Efficient information retrieval in peer-to-peer networks. Technical report, HP Labs, 2002.

[202] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB'04*, pages 648–659, 2004.

[203] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB'03*, pages 333–344, 2003.

[204] D. A. Tran. Hierarchical Semantic Overlay Approach to P2P Similarity Search. In *USENIX Technical Conference*, pages 355–358, 2005.

[205] Travel-and-guide homepage. http://www.travel-and-guide.com (last visited 2009/02/21).

[206] P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *Int. Workshop On Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'04)*, pages 169–183, 2004.

[207] W. Vogels. Data access patterns in the Amazon.com technology platform. In *VLDB'07*, pages 1–1, 2007. Keynote talk, video available at http://www.vldb2007.org/program/keynotes.html (last visited 2009/02/21).

[208] S. Wang, B. Ch. Ooi, and A. K. H. Tung. Efficient skyline query processing on peer-to-peer networks. In *ICDE'07*, pages 1126–1135, 2007.

[209] S. Wang, Q. H. Vu, B. Ch. Ooi, A. K. H. Tung, and L. Xu. Skyframe: a framework for skyline query processing in peer-to-peer systems. *The VLDB Journal*, 18(1):345–362, 2009.

[210] M. Wiegandt. Ähnlichkeitsanfragen in P-Grid-basierten P2P-Systemen, 2006. Diploma Thesis at TU Ilmenau (available in German only).

[211] Wikipedia homepage, 2001. http://www.wikipedia.org (last visited 2009/02/21).

[212] W. Wirth. Datenbankintegration für virtuelle Observatorien, 2004. Diploma Thesis at TU Ilmenau (available in German only).

[213] P. Wu, C. Zhan, Y. Feng, B. Zhao, D. Agrawal, and A. El Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT'06*, pages 112–130, 2006.

[214] S. Wu, J. Li, B. Ch. Ooi, and K.-L. Tan. Just-in-time query retrieval over partially indexed data on structured P2P overlays. In *SIGMOD'08*, pages 279–290, 2008.

[215] Y. Xiong, X. Lin, and J. A. Rowson. Estimating Device Availability in Pervasive Peer-to-Peer Environment. In *Int. Workshop on Future Trends of Distributed Computing Systems (FTDCS'04)*, pages 254–260, 2004.

[216] XML at W3C homepage. http://www.w3.org/XML (last visited 2009/02/21).

[217] C. T. Yu and W. Meng. *Principles of database query processing for advanced applications.* Morgan Kaufmann Publishers Inc., 1998.

[218] M. Zaharia, A. Chandel, S. Saroiu, and S. Keshav. Finding Content in File-Sharing Networks When You Can't Even Spell. In *Int. Workshop on P2P Systems (IPTPS'07)*, 2007.

[219] Zattoo homepage. http://zattoo.com (last visited 2008/11/12).

[220] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

About the Author

Marcel Karnstedt was born in 1977 in Halle (Saale), Germany. He received his Diploma (M.Sc.) from the Martin-Luther-Universität Halle-Wittenberg, Germany in December 2003. From January 2004 to February 2009 he had been working as a research associate and teaching assistant with the Databases & Information Systems Group (head: Prof. Dr. habil. Kai-Uwe Sattler) at the Ilmenau University of Technology, Germany. That is where he also successfully completed his Ph.D. (summa cum laude) on the topic of P2P databases. During his time in Ilmenau he was working also in the area of data mining with a focus on data streams. Further, he led more than 20 labs, seminars, practical courses and successfully taught a course on database implementation techniques for Diploma (M.Sc.) students. He had been actively involved in supervising several Diploma theses and interns.

Since March 2009 he has been employed as a Postdoctoral Researcher at the Digital Enterprise Research Institute (DERI), National University of Ireland, Galway (NUIG). He is member of the Unit for Information Mining and Retrieval (UIMR) and participates in the CLIQUE project on analyzing and visualizing large graphs and networks, specifically social networks and biological networks. His current work at DERI concerns the development of intelligent knowledge management techniques for on-line communities, particularly focusing on cross-community effects as well as data management, provisioning and processing issues. Further, he is involved in ongoing research in the areas of databases and data management in general.

At the time of this book, Marcel Karnstedt has published more than 30 articles in journals and refereed conference and workshop contributions. He has presented his work at international conferences and workshops such as the International Conference on Data Engineering (ICDE), the International Conference on Information and Knowledge Management (CIKM), and the IEEE Conference on Peer-to-Peer Computing (P2P). Currently, he acts as a reviewer for several international conferences and journals and gives invited talks, such as at the IFIP 2.6 - Research Workshop 2009. He further holds three IBM certificates in database use and administration (DB2 Universal Database V8.1 for Linux UNIX and Windows).

For further details please visit http://www.marcel.karnstedt.com.