Keyword Search across Distributed Heterogeneous Structured Data Sources

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von: Dipl.-Inf. Ingolf Geist
geb. am 21. Januar 1975 in Wernigerode

Gutachter:
Prof. Dr. rer. nat. habil. Gunter Saake
Prof. Dr.-Ing. habil. Kai-Uwe Sattler
Prof. Dr. rer. nat. habil. Wolfgang May

Ort und Datum des Promotionskolloquiums Magdeburg, 14. September 2012

Geist, Ingolf:
Keyword Search across Distributed Heterogeneous Structured Data Sources
Dissertation, Otto-von-Guericke-Universität Magdeburg, 2012.

Abstract

Many applications and users require integrated data from multiple, distributed, heterogeneous (semi-) structured sources. Sources are relational databases, XML databases, or even structured Web resources. Mediator systems represent one class of solutions for data integration. They provide a uniform view and uniform way to query the virtually integrated data. As data resides in the local sources, global queries are translated into source queries, and the obtained local results are combined to be presented to the users or global applications. Motivated by Semantic Web ideas, concept-based mediator systems have emerged. This class of systems describes global data in the form of a domain-specific concept schema, taxonomy, or even an ontology. In this way, they allow the connection of structurally different data by semantic relationships. The mediator systems provide the mapping of the local data to the global concept-schema and the query planning and processing. Most existing systems provide complex concept-based query languages as query interface. The systems are hard to use for global users because of their incomplete knowledge about the concept schema, the data, and the query language.

At the same time, keyword search has become ubiquitous and successful in Internet and document search. Simple keyword queries allow the fast retrieval of relevant Web pages and documents. Dealing with (semi-)structured data causes new challenges for keyword search. There is a large body of research for keyword search over relational databases, XML trees and graphs, and general graph data. The main problem is that information in structured databases is distributed over different connected objects like tuples, attribute values, or XML elements. The search systems have to find connected trees, sub-graphs, or networks that answer a keyword query together. There are two main approaches. Schema graph-based approaches generate candidate queries based on keyword positions initially, e.g., join queries, and execute the queries to obtain the actual answers in a second step. Data graph-based systems search directly in the data graph for connected objects that answer the keyword query. Most systems require a centralized database or non-restricted query interfaces. In this thesis, we have the goal to allow keyword search over distributed, multiple, heterogeneous, semi-structured data sources.

For this task, we combine a concept-based mediator system with a schema graph-based keyword search approach. We use and extend the concept-based mediator system YACOB. The system provides a concept-based schema and allows the uniform concept-based querying of relational databases, XML databases, and structured Web sources. It is used to answer queries over single concepts, while the keyword search systems finds semantically connected objects that answer a keyword query. For this, the keyword search system uses a global index that contains keywords and information in which concepts and properties they occur. The index does not contain object identifiers avoiding a complete materialization on the global level. In the thesis, we

define the semantics of keyword queries in a concept-based data model. We introduce concept-based query terms that are an instance of labeled keyword queries. In the following step, concept-based queries as interpretation of keywords and their ranking are presented. Query expansion exploiting the concept-based schema adds flexibility to labeled keyword terms.

For the generation of object networks as results, we follow the schema graph-based approach, because we assume we do not have a materialized data graph. Therefore, we propose and validate algorithms for efficient query generation in complex concept-schema graphs. In the second step, we investigate the efficient execution of the candidate queries. We propose a semi-join and bind-join based approach for joining objects from different sources. The approach reduces the number of materialized objects that have to be transferred to and stored on the global level. We show that many queries generated by keyword queries overlap. Hence, we propose and validate methods for reusing intermediate results in different queries and for the detection of empty results to improve existing algorithms for centralized databases. Validation experiments showed that both materialization of intermediate results and empty-result detection are necessary and significantly improve the performance.

Zusammenfassung

Durch die Entwicklung der Computer- und Netzwerktechnologien sowie von Informations- und Datenbankmanagementsysteme steht heute eine Vielzahl von strukturierten Datenquellen zur Verfügung. Diese Datenquellen können relationale oder XML-basierte Datenbanksysteme oder einfache, strukturierte Web-Datenbanken sein. Es existieren viele Anwendungen, die zur Erfüllung des Informationsbedürfnisses nicht nur auf eine Quelle zurückgreifen können, sondern integrierte Daten und Informationen aus verschiedenen, verteilten, heterogenen strukturierten Datenquellen benötigen. Mediatoren stellen eine Lösung dar, um Anwendungen eine solche integrierte Sicht zu ermöglichen. Mediatoren sind eine Middleware-Lösung, die eine einheitliche Sicht und eine einheitliche Anfragesprache für lokale Quellen bereitstellt, wobei die Daten in den lokalen Systemen verbleiben. Mit der Entwicklung von Semantic-Web-Techniken kamen konzeptbasierte Mediatorsysteme auf. Auf globaler Ebene wird die Applikationsdomäne mit Hilfe eines Konzeptschemas, eines Vokabulars, oder einer Ontologie beschrieben. Die globalen Applikationen und Benutzer nutzen diese Sicht, während die Mediatorsysteme die Abbildung von lokalen Typen und Daten auf das globale Konzeptmodell und -schema bereitstellen, sowie die Anfrageplanung und -ausführung als auch die Integration übernehmen. Die konzeptbasierte Beschreibung und Anfrageverarbeitung hat den Vorteil, dass strukturell unterschiedliche aber semantisch verwandte Informationen leicht über semantische Beziehungen verbunden werden können. Konzeptbasierte Mediatorsysteme stellen eine komplexe Anfragesprache bereit. Zusammen mit einem unvollständigen Wissen über das Konzeptschema führt das zu Schwierigkeiten bei der Nutzung der integrierten Daten.

In den letzten Jahren stellt Stichwortsuche im Internet und Dokumentenbeständen den Benutzern eine einfache und erfolgreiche Schnittstelle bereit. Somit wurde das natürliche Ziel aufgestellt, Stichwortsuche für (semi-)strukturierte Datenbanken zu ermöglichen. Durch die Verteilung von Informationen auf unterschiedliche Objekte, zum Beispiel Tupel und Attributwerte oder XML-Elemente, wurde das Problem der Stichwortsuche auf das Finden von verbundenen Objekten ausgeweitet, die zusammen eine Antwort auf die Stichwortanfrage bilden. Es existieren viele Ansätze für die Suche über relationale Datenbanken, XML-Bäume und -Graphen als auch über allgemeine Datengraphen. Diese Ansätze können in zwei Klassen unterteilt werden: dem schemagraph-basierten und dem datengraph-basierten Ansatz. Der schemagraph-basierte Ansatz erstellt zunächst strukturierte Anfragen, beispielsweise Verbundanfragen, aus den Stichwortpositionen bezüglich des Schemas. Diese Anfragen werden anschließend ausgeführt, um die Antworten auf die Stichwortanfrage zu erstellen. Datengraph-basierte Ansätze modellieren die Daten direkt als Graph und arbeiten direkt auf diesem, um Antworten auf die Stichwortanfrage zu generieren. Die meisten dieser Systeme erfordern entweder ein zentralisiertes Datenbanksystem oder zumindest eine uneingeschränkte SQL-Anfrageschnittstelle. In dieser Schrift stellen wir uns das

Ziel, effiziente Stichwortsuche über verteilte, heterogene, (semi-)strukturierte Datenquellen zu ermöglichen.

Zur Lösung dieser Aufgabe kombinieren wir das konzeptbasierte Mediatorsystem YACOB mit einem schemagraph-basierten Stichwortsuchansatz. Zunächst wird die Semantik von Stichwortanfragen definiert. Dabei wird zwischen puren Stichworten und konzeptbasierten Anfragetermen, einer Instanz von Stichworten mit Label, unterschieden. Wir beschreiben die Interpretation von Stichwortanfragen als konzeptbasierte Materialisierungsanfragen mit Hilfe eines Stichwortindexes. Der Index beschreibt die Position von Stichworten bzgl. Konzept, Property und Datenquellen, aber nutzt keine globale Objekt-Identifier, um eine komplette Materialisierung der Daten auf globaler Ebene zu vermeiden. Für top-k Anfragen wird eine monotone Rankingfunktion erarbeitet, die Materialisierungsanfragen hinsichtlich ihre Relevanz einordnet. Eine Anfrageerweiterung für konzeptbasierte Anfrageterme ermöglicht die flexible Nutzung von Konzeptvorschlägen durch den Benutzer.

Für die Beantwortung von Stichwortanfragen nutzen wir den schemagraph-basierten Ansatz, da kein globaler, materialisierter Graph von Instanzen vorhanden ist oder erstellt werden soll. Wir stellen einen effizienten Ansatz vor, um konzeptbasierte Anfragen zu generieren. Anschließend untersuchen wir die effiziente Ausführung von konzeptbasierten Anfragen. Dabei schlagen wir einen Ansatz vor, der Semiverbund- und Abhängigkeitsverbundoperationen (Bind-Join) zur Erstellung von Objektnetzwerken benutzt. Dieses ermöglicht die Nutzung von Quellen mit limitierten Anfrageschnittstellen und reduziert die Menge der materialisierten Objekte auf globaler Ebene. Wir zeigen, dass sich die generierten Anfragen für eine Stichwortmenge stark ähneln, deswegen schlagen wir vor, Zwischenergebnisse zwischen Anfragen zu teilen. Wir nutzen dabei sowohl Methoden zur Erkennung von leeren Ergebnissen als auch einen semantischen Zwischenspeicher. Mit Hilfe einer prototypischen Umsetzung validieren wir unsere Ansätze und zeigen, dass die Kombination des Erkennens von leeren Anfragen mit dem Cacheansatz die besten Ergebnisse ermöglicht und die Performance deutlich steigert.

Acknowledgments

It took me a long time to finish this thesis. During this time, I have met great colleagues and learned many different things, also outside of this topic. I took part in several projects that provided new challenges. I am truly thankful that I was allowed to study and work at the University of Magdeburg, at the Department of Computer Science, and in the research group of Prof. Gunter Saake. Prof. Saake started my interest in database technologies with his Database II lecture. After I joined the research group, Prof. Saake gave me always freedom for my research. His advices were immensely helpful and welcome. I want to thank my third reviewer Prof. Wolfgang May. He gave me extremely helpful and detailed comments to a preliminary version of this thesis, which helped me to improve the thesis significantly. My second reviewer Prof. Kai-Uwe Sattler was responsible for joining me the database research group. At first, I was his student research assistant, then I wrote my diploma thesis under his guidance. Later we collaborated in several papers about Information Fusion, Database Self-tuning, and Concept-based mediator systems. Prof. Sattler created the concept-based mediator system Yacob, which is the foundation of this thesis. The students Torsten Declercq, Marcel Karnstedt, and Reiner Habrecht took part in this development, too. Prof. Sattler read many versions of this thesis and provided helpful comments to improve this thesis.

I enjoyed my work in the research group very much. I liked the atmosphere and all the members. In particular, I have to thank my colleague Dr. Eike Schallehn. We shared an office, talked a lot about the work, music, and life. We collaborated in papers about similarity joins in virtual data integration environments and database self-tuning. Eike read all chapter of my thesis, gave many helpful comments. Furthermore, he became a friend. I want to thank my other office mates Hagen Höpfner, Anke Schneidewind, and Andreas Lübcke. I want to thank all persons who read parts of the thesis. It helped me a lot and reassured me about my thesis. I want to thank Ateeq Khan, Andreas Lübcke, Sebastian Bayer, Sebastian Breß, and Angela Brennecke.

Finally, I want to thank the persons in my personal life. My parents always believed in me and supported me in many directions. Without them, everything would have be much harder in my life. I want to thank my friends who helped me in many ways to finish my thesis. I want to thank my great friend Christiane who has been very important to me during the last 12 years. I do not know what to do without her. I want to thank my partner Mareike. She gave me motivation and stability in the last phase of the thesis.

Contents

Contents vii

List of Figures xi

List of Tables xv

1. Introduction 1
- 1.1. Contributions . . . 3
- 1.2. Structure . . . 5

2. Data Integration 7
- 2.1. Data Integration Systems . . . 7
  - 2.1.1. Challenges and Classification of Data Integration Systems . . . 7
  - 2.1.2. Mediator Systems . . . 11
- 2.2. Concept-based Mediator Systems . . . 15
  - 2.2.1. Principles . . . 16
  - 2.2.2. Approaches and Systems . . . 18
  - 2.2.3. Other approaches . . . 20
- 2.3. Summary . . . 21

3. Yacob 25
- 3.1. Integration Model . . . 26
  - 3.1.1. Concept model and Data model . . . 26
  - 3.1.2. Source mappings . . . 29
- 3.2. Query Language . . . 32
- 3.3. Query Processing . . . 39
- 3.4. Summary . . . 41

4. Keyword Search in Structured Databases 43
- 4.1. General Considerations . . . 43
  - 4.1.1. Example . . . 44
  - 4.1.2. Definitions . . . 46
- 4.2. Classification . . . 48
  - 4.2.1. Query Languages . . . 48
  - 4.2.2. Query Result Types . . . 49
  - 4.2.3. Ranking Methods . . . 53
  - 4.2.4. Query Processing in Keyword Search Systems . . . 60
- 4.3. Keyword Search Systems . . . 62
  - 4.3.1. Relational Database Search . . . 62

4.3.2. Keyword Search in XML Trees and Graphs 64
4.3.3. Relaxed Structural Queries 66
4.3.4. XML Query Languages and Keyword Search 69
4.4. Keyword Search and Virtual Integrated Data Sources 72
4.4.1. Metasearch Engines 73
4.4.2. Structured Hidden-Web Search 75
4.4.3. Virtual Integrated Structured Sources 77
4.5. Summary 83

5. Concept-based Keyword Queries 85
5.1. Motivation 86
5.2. Data Model 89
5.3. Query Model 93
5.3.1. Keyword Queries 93
5.3.2. Minimal Object Networks 94
5.3.3. Materialization Queries 96
5.3.4. Materialization Query Scoring Function 102
5.4. Query Expansion 106
5.4.1. Semantic Distance of Classes 106
5.4.2. Concept Expansion 108
5.4.3. Category Expansion 109
5.5. Related Work 110
5.6. Summary 112

6. Concept Query Generation 113
6.1. Overview 114
6.2. Keyword Processor 116
6.2.1. Keyword Index 116
6.2.2. Index Lookup Methods 119
6.3. Query List Generation 122
6.3.1. Index List Generation 122
6.3.2. Single Concept Query Generation 123
6.3.3. Concept Schema Graph Annotation 126
6.4. Query List Network Enumeration 126
6.4.1. Query List Network 127
6.4.2. Compact Concept Schema Graph 128
6.4.3. Enumeration of Query List Networks 130
6.5. Discussion 132
6.6. Summary 133

7. Concept Query Processing 135
7.1. Preliminaries 136
7.1.1. Query Processing 136
7.1.2. Non-Reuse Algorithms 146
7.1.3. Motivation for Query Re-Using Algorithms 148

7.2. Detection of Empty Results 149
7.2.1. Concept-based Query Coverage 151
7.2.2. Data Structure 153
7.2.3. Empty Result Statistics Management 154
7.2.4. Use of Detected Empty Results during Keyword Query Processing 155
7.3. Re-Using Intermediate Results 156
7.3.1. Materialized Results Organization and Access 156
7.3.2. Cache Management 165
7.3.3. Single Query Cache Use 166
7.4. Optimizing Query Result Reuse 168
7.4.1. Query List Network Optimization 168
7.4.2. Query List Network-based Query Reusing 171
7.5. Query List Optimizations 173
7.5.1. Splitting Query Lists 174
7.5.2. Merging Single Concept Queries 174
7.6. Discussion 175
7.7. Summary 179

8. Implementation and Evaluation 181
8.1. Architecture and Implementation 181
8.2. Scenarios and Evaluation Measures 185
8.2.1. Evaluation Goals 185
8.2.2. Data Sets and Query Sets 187
8.2.3. Evaluation Environment 188
8.3. Efficiency Evaluation 189
8.3.1. Size estimation evaluation 189
8.3.2. Detailed Evaluation 191
8.3.3. Top-k evaluation 196
8.4. Effectiveness Evaluation 201
8.5. Summary 203

9. Summary and Future Work 205
9.1. Summary 206
9.2. Contributions 207
9.3. Future work 208

Bibliography 211

A. Data and Query Sets 239
A.1. Data Set 239
A.1.1. Concept schema IMDB 239
A.2. Query Sets 240
A.2.1. Queries Cost Estimation 240
A.2.2. Enumeration Test Queries 243
A.2.3. Query execution experiment 244
A.2.4. Top-k query sets 245

A.2.5. Concept-based query experiment 245
A.2.6. Effectivness Experiment Results 248

List of Figures

1.1. Initial Situation . . . 2
1.2. Goal: Keyword search system . . . 4
1.3. Schematic overview over the contributions and challenges . . . 5

2.1. Classification of systems for Data Integration . . . 11
2.2. Mediator architecture . . . 13
2.3. Mediator query processing . . . 14

3.1. Overview: Integration . . . 25
3.2. Exemplary Yacob concept schema . . . 27
3.3. Exemplary Yacob mapping . . . 31
3.4. Listing CQuery query Q_1 . . . 32
3.5. Listing CQuery query Q_2 . . . 34
3.6. Listing CQuery query Q_3 . . . 34
3.7. Listing CQuery query Q_4 . . . 34
3.8. Listing CQuery query Q_5 . . . 39

4.1. Exemplary structured database . . . 45
4.2. Schema graph example . . . 47
4.3. Schema graph-based vs. data graph-based keyword search . . . 60
4.4. Structural relaxation . . . 68
4.5. Combination of precise querying and IR . . . 69
4.6. Example query XIRQL . . . 70
4.7. Example query XXL . . . 70
4.8. Example query XQuery Full-Text . . . 71
4.9. Metasearch engine components . . . 73
4.10. KITE system architecture . . . 78
4.11. Querying over virtual XML views . . . 79
4.12. SINGAPORE data space . . . 82

5.1. Overview keyword semantics . . . 85
5.2. Concept Schema Example . . . 87
5.3. Exemplary objects . . . 87
5.4. Listing CQuery keyword example . . . 88
5.5. Virtual document . . . 90
5.6. Exemplary minimal object networks . . . 95
5.7. Listing materialization query translation result . . . 101
5.8. Illustration of Semantic Distance . . . 107
5.9. Category Expansion Example . . . 110

6.1. Overview: concept query generation 113
6.2. Keyword query processing steps 114
6.3. Exemplary annotated concept graph 115
6.4. Exemplary query list networks 116
6.5. Offline phase of the keyword search system 117
6.6. Dewey numbering 118
6.7. Index entry structure 118
6.8. Index example 120
6.9. Single query term lookup 120
6.10. Query list generation 124
6.11. Concept graph annotation 127
6.12. Example graph and corresponding annotated graph 129
6.13. Annotated compact example graph 130

7.1. System overview: Concept query processing 135
7.2. Exemplary query networks 137
7.3. Plan operator 139
7.4. Execution plans 144
7.5. Query list network 147
7.6. Query list network processing 148
7.7. Query overlapping 150
7.8. Empty result set detection 153
7.9. Empty result representation 154
7.10. Cache structure 160
7.11. Exemplary distinct cases 160
7.12. Example plan for cache usage 166
7.13. Example query processing 169
7.14. Shared query list networks 172

8.1. System architecture 182
8.2. Estimations and actual values for query set 9 190
8.3. Estimated values vs. actual values 191
8.4. Query list generation times 192
8.5. Number of generated single concept queries and query lists 193
8.6. Query list enumeration: $size_{max}$ vs. time 194
8.7. Query list enumeration: keyword query size $|Q|$ vs. time 194
8.8. Influence of the keyword occurrence $KWOcc$ on the execution time . . 195
8.9. Execution times with respect to the parameter $size_{max}$ for $|Q| = 3$ . . . 196
8.10. Optimization vs. execution time 197
8.11. Optimization strategies vs. execution time 198
8.12. Top-k overall execution times for different $size_{max}$ and $|Q| = 2$ 199
8.13. Top-k overall execution times for different keyword sizes 199
8.14. Top-k query list splitting 200
8.15. Top-k complete execution times of concept-based keyword queries . . . 201
8.16. Top-k complete execution time vs. concept-based keyword size 201

A.1. IMDB concept schema 239
A.2. Result of query 1 248
A.3. Result of query 2 249
A.4. Result of query 3 249
A.5. Result of query 4 250
A.6. Result of query 5 250
A.7. Result of query 6 251
A.8. Result of query 7 252
A.9. Result of query 8 253
A.10. Result of query 9 254
A.11. Result of query 10 254
A.12. Result of query 11 255

List of Tables

6.1. Index term types 119

7.1. Statistics values 139
7.2. Plan operators 140
7.3. Percentage of overlapping query parts (single concept queries/joins) . . 149
7.4. Match types in the semantic cache 161

8.1. Keyword index schema 183
8.2. Main evaluation parameters 187
8.3. Database structure 188
8.4. IMDB sources 188
8.5. Query set characteristics 189
8.6. Estimation errors 190
8.7. Compared algorithms 195
8.8. Query set for the effectiveness experiment 202

1. Introduction

The development of computer and network technologies and information systems has significantly improved the availability of structured data sources. As every source is developed and managed with local applications in mind, the sources differ in design, data model, query capabilities and interface, and data representation. Nevertheless, many applications of large enterprises, in interdisciplinary research, or in Web search and commerce do not only require one information source, but need integrated information from multiple, distributed, heterogeneous and autonomous structured sources. Ideally, global applications and users expect a uniform view of and a transparent access to the data sources. This situation makes data integration a pervasive task [HRO06]. Data integration systems provide a structured or semi-structured interface to data. That means that they offer a schema and a corresponding query language. For this task, integration systems provide a global schema. Integration systems use mappings between the local schemas and the global schema to rewrite global queries and integrate the results. First generation integration systems use a structural integration model. That means that the global application domain is modeled using a structural database model. Subsequently, local schema elements are mapped to the global schema elements. However, this leads to complicated, global schemas [SGS05]. Furthermore, it is difficult to connect sources that are semantically related but do not have a structural overlapping [LGM01].

Following the Semantic Web, an alternative is explicit modeling and use of the application domain in the form of taxonomies, concept hierarchies, or an ontology [AK93, WVV+01, LGM01, ABFS02a, SGS05, LWB08]. With the help of explicit domain modeling, different worlds can be connected, and the concept schema is simplified [LGM01]. To use semantic modeling in data integration, we still need a middleware or mediator to map local sources to the global concept-based schema and rewrite global queries against the concept-based schema to local queries. For example, Figure 1.1 sketches this approach. The user formulates a query that is expressed in a concept-based query language. The mediator system processes the query, creates a query plan and selects sources. Subsequently, the system sends queries to the data sources via wrappers and retrieves the results. The result objects are transformed into instances of the global concept model and returned to the user.

While explicit domain knowledge eases querying of the integrated database, the following problems still exists [JCE+07]:

1. The users are not entirely familiar with the domain specific ontology that represents the data structure. The users do not know all relationships or properties. Furthermore, users do not have knowledge about the global vocabulary, but they know the structure of single sources. They cannot use the global system efficiently.

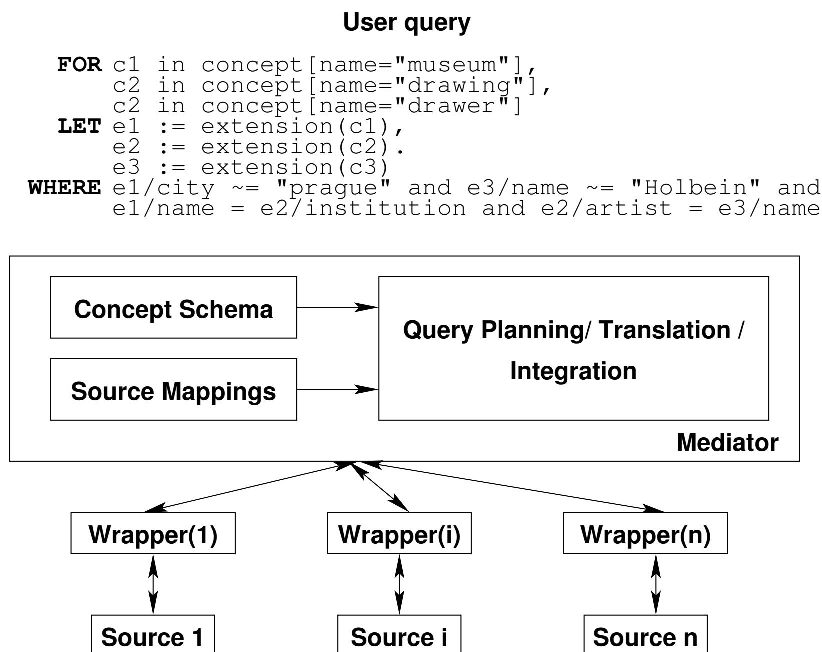


Figure 1.1.: Initial Situation

2. The users are not experts in using complex query languages. Concept-based query languages are based on ontology languages, are RDF languages like SPARQL, or are derivatives of OQL, XQuery, etc. They are hard to use. Query forms and browsing mitigate this problem but restrict users to predefined queries.

3. Information is distributed over different global data elements and even sources. The user cannot know exactly, which relationships she should query. As data comes from different sources, this problem is even increased.

In summary, we need a way to provide an easy to use interface for querying across multiple heterogeneous structured data sources using a mediator system.

At the same time, easy-to-use query interfaces like keyword search are successful for unstructured data and in Web search. Users provide simply a set of keywords and the systems return a list of ranked documents. In the recent decade, the database research community has noticed the necessity of combining database technology and information retrieval [CRW05]. An important point in this consideration is the usability of databases [JCE+07]. Keyword search is one solution. Information is spread across different connected data items (tuples, XML elements, etc.) in (semi-)structured databases. This leads to the problem of finding connected data item that satisfy the keyword query [YQC10]. Research prototypes and industrial systems exist for relational databases [HGP03, ACD02, LYMC06, LLWZ07, WZP+07, LFZ08, QYC09, PgL11, LFZW11], XML trees and graphs [FKM00, GSBS03, CMKS03, BHK+03, XP08], and general data graphs [DEGP98b, HN02, HWYY07, LOF+08, LFZ08, LFO+11]. We can distinguish two main keyword processing approaches: schema graph-based and data graph-based algorithms. Schema graph-based approaches create candidate queries first, for exam-

ple, join queries via foreign-key relationships. In the second step, they execute the generated queries to obtain the results. In contrast, data graph-based approaches represent the data items and the keywords as nodes in a graph, where nodes are connected by edges. Results are trees or sub-graphs that connect the keyword nodes.

While there are meta search systems [MYL02] and deep-web search systems [CHZ05], there are only few proposals that consider keyword search across multiple, heterogeneous, and partially autonomous structured sources, e.g., [SLDG07, SGB+07]. In this thesis, we combine concept-based integration and keyword search to allow keyword queries over concept-based, virtually integrated data.

We assume that we have an available concept-based mediator system Yacob [SGHS03, SGS05]. Furthermore, we require content descriptions of every source, i.e., a list of keywords and their positions with respect to the global concept schema. We develop a *schema graph-based keyword search system* over the concept-based data model. Given a keyword query, the system generates concept-based queries creating connected objects, i.e., object networks, that contain all keywords. The generation and execution of these materialization queries are expensive operations. In particular, we assume limited query capabilities of the sources, e.g., Web sources. Thus, we need to develop efficient generation and execution strategies for sets of materialization queries. The goal of the thesis is a keyword search system as depicted in Figure 1.2. The user formulates a keyword query and sends it to the keyword search system. The system creates concept-based queries, executes them, and returns relevant object networks to the user. It uses a keyword index to find keyword interpretations. A query generation component combines interpretations to materialization queries. The execution component efficiently executes the queries and ranks the results.

1.1. Contributions

To improve keyword search across multiple heterogeneous structured data sources, we contribute following points:

Concept-based keyword search: The basis of the keyword search is a concept-based integration model. We define the keyword search semantics of this data model. In particular, we specify concept-based query terms that use concept and property names as labels. Thereby, we also consider information from source mappings to reflect the heterogeneous nature of an integration system. We provide a ranking function for queries that are generated from keyword queries. We exploit the concept schema to expand labeled keywords in order to allow inexact label matchings.

Efficient candidate query generation: Our goal is to develop a concept-based and schema graph-based keyword search system. In the first step, concept-based queries are generated from keyword queries. Concept schema graphs are complex. This problem is increased by concept hierarchies that multiply connections between concepts. Therefore, we reduce the complexity by collapsing concept hierarchies and edges into complex schema nodes and edges. This significantly reduces the graph complexity and allows the efficient generation of concept-based

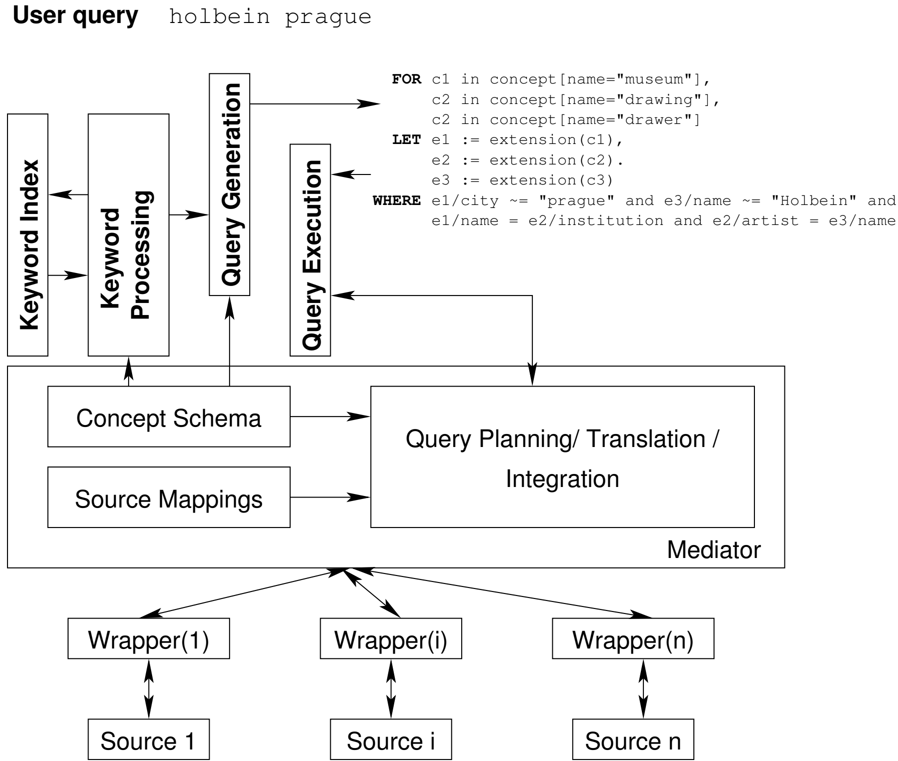


Figure 1.2.: Goal: Keyword search system

queries to materialize results. The compact concept graph approach extends similar approaches for relational and XML data.

Efficient execution of candidate queries: The schema graph-based approach generates many queries as interpretations of keywords. We argue that many queries overlap, i.e., intermediate results can be reused. The execution of source queries is expensive because of network access and limited query capabilities. There are two ways of sharing intermediate query results in materialization queries: *empty result detection* and *materialization of intermediate results*. In the first case, we avoid unnecessary re-computation of empty results. In the second case, we avoid re-computation of intermediate results. We provide definitions of query coverage and query containment for concept-based queries. We propose different data structures that hold descriptions of empty query results and provide a semantic cache. At last, we provide an approach to combine the plans of different materialization queries and to optimize them together.

For the execution of query sets, we develop a join approach based on semi-joins. In this approach, we materialize all candidate objects, first, and combine the results, subsequently. In summary, we optimize concept-based query processing with respect to keyword queries.

Prototype and evaluation: We develop a prototype to validate the developed concepts. The prototype uses the YACOB mediator system to access sources. Additional components comprise the keyword index, the query generator as well

as the query executor, and the join processor. We validate all presented approaches with the help of a freely accessible dataset and several representative queries sets. The prototype provides the validation that the schema graph-based keyword search approach across heterogeneous and autonomous sources via a concept-based mediator is possible.

The contributions are summarized in Figure 1.3. The thesis is structured according these tasks.

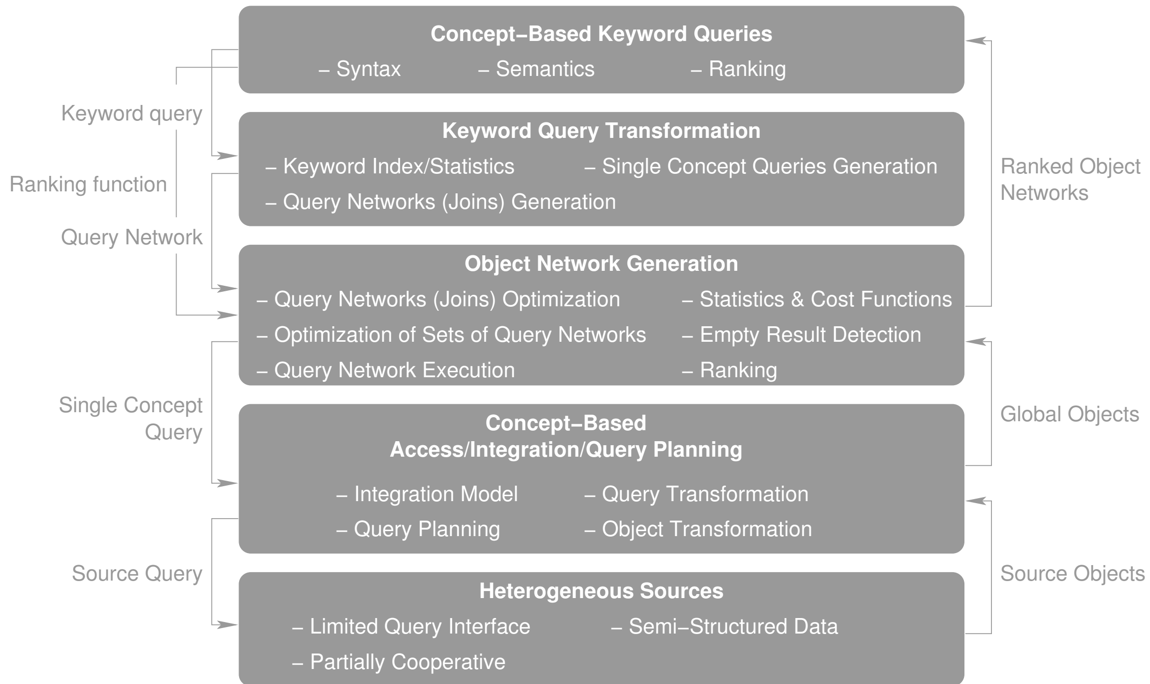


Figure 1.3.: Schematic overview over the contributions and challenges

1.2. Structure

The remainder of the thesis is structured as follows:

Chapter 2 discusses the background of data integration systems. We focus on concept-based mediator systems.

Chapter 3 describes the YACOB mediator system [SGHS03, KSGH03, SGS05]. The concept-based mediator is the basis of the keyword search system. We provide the integration model, the concept-based query language, and the description of the query processing. This chapter shares parts with [SGHS03, KSGH03, SGS05].

Chapter 4 reviews keyword search in (semi-)structured databases. We describe the principles of keyword search in (semi-)structured databases and focus in the second part of the chapter on keyword search across multiple heterogeneous structured sources.

Chapter 5 defines concept-based keyword queries, the data model for keyword search, and the ranking function. We define query expansion for labeled keyword queries. This chapter and the following chapters are partially based on and extend [GDSS03, Gei04, Dec04].

Chapter 6 describes the complete concept schema-graph based keyword search process. The chapter focuses on query generation from concept-based keyword queries.

Chapter 7 deals with the efficient execution of a set of generated structured queries. The results of these queries form the result of the keyword search. Thereby, the system exploits overlapping queries to reduce query costs.

Chapter 8 includes the description of the prototypical implementation and the evaluation of the system to validate the approach.

Chapter 9 concludes the thesis, summarizes the results, and provides a list of possible future steps.

2. Data Integration

For several decades, many projects in industry and research have dealt with the problems of the integration of autonomous, distributed, and heterogeneous data sources. The sources are available on Web pages, via Web services, or by other protocols. In particular, the ubiquitous Internet makes many sources to be available for integration. Thereby, there is still the necessity to integrate data from different sources in many applications areas. Data integration is a challenging task that requires a solution for many problems like resolving integration conflicts caused by autonomy and heterogeneity of the different sources, overcoming technical problems introduced by heterogeneous system architectures, and optimization of query processing in distributed data integration systems.

In this chapter, we give an overview of data integration systems and we also include former surveys [DD99, Con97, Hul97, BKLW99, SL90, She99, HRO06]. In Section 2.1), we describe data integration systems in general. First, we classify the challenges of data integration as well as data integration systems. Second, we investigate mediator systems in detail. In Section 2.2, we review *Concept-based Mediator systems* and their designs as one kind of data integration systems. We conclude the chapter with a summary and a discussion of the presentation level of concept-based mediator systems in Section 2.3.

2.1. Data Integration Systems

We start this section with the description of challenges of data integration systems caused by multiple autonomous and heterogeneous sources. Following this, we turn to mediator systems as one exemplary class of data integration systems to deal with the given challenges.

2.1.1. Challenges and Classification of Data Integration Systems

Sheth and Larsen [SL90] proposed a classification of information systems along the dimensions *Autonomy*, *Heterogeneity*, and *Distribution* that is also used in following surveys [Con97, BKLW99]. The classification also describes the main problems of information integration systems. Although, these classification dimensions were developed 20 years ago, they still show the problems of systems today. We now discuss the influence of the three dimensions on data integration in detail.

Autonomy

The first classification dimension describes the extent of autonomy of the data sources to be integrated. We can distinguish design, communication, and execution autonomy. Design autonomy comprises independence of design and independence of design change. Users and applications require the independent design of the local databases. Furthermore, the local schemata and information representations can change at any point of time. A source can always decide if it communicates with external programs or not. That kind of autonomy is called communication autonomy. Execution autonomy states that the local execution of queries cannot be influenced by a global site. The execution is only based on local decisions. The integration of Web sources increases this problem because of the manifold of the sources and the limited access and control of them.

Heterogeneity

Autonomous development, usage, and design of the local information systems introduce heterogeneity between the systems. The heterogeneity concerns different levels. Several studies describe different levels of heterogeneity [Wie93, Con97, BKLW99]. The heterogeneity is characterized as follows:

Syntactical or technical heterogeneity: The first class of heterogeneity comprises technical inconsistency like different operating systems, database systems, and protocols. Furthermore, there is a variety of access and security methods. Systems provide own interfaces to their data with different capabilities. The possibilities range from full-featured database management systems with powerful query languages as SQL or XQuery to Web information systems that provide an HTML form or a Web service interface as query interface.

Data model heterogeneity: Local database systems support their own data model. Exemplary data models are the relational, object-oriented, or semi-structured XML model. Different semantics of local data models induce heterogeneity on data model level. A data integration system has to deal with data model heterogeneity by translating the local data models into the global data model. It is reflected in the 5-layer schema architecture of Federated Database Systems [SL90] as well as in the wrapper layer in the Mediator-based architecture [Wie92].

Logical heterogeneity: The logical heterogeneity concerns schematic and semantic conflicts.

Schematic heterogeneity is caused by design autonomy, which leads to different schemas for same real world concepts even if a common data model is used. For example, in the relational model differences are generated by using different levels of normalization. Even more classes of conflicts can occur in semantically rich data models [Sch98].

Metadata conflicts are a significant class of conflicts caused by schematic heterogeneity. For example, one source models information as metadata, e.g., attribute or relation names, another source represents the same real

world information as data values. The solution of these conflicts requires query languages of higher order like SchemaSQL [LSS96], MSQL [KLK91], or FraQL [SCS03].

Several surveys provide comprehensive overviews about integration conflicts. For example, Kim and Seo investigate conflicts between relational databases [KS91]. Schmitt compares different classification approaches [Sch98], while Conrad proposes a further classification [Con97].

Semantic heterogeneity: Even if a common data structure is used, semantic conflicts can occur. For instance, in the XML data model, different tags carry implicit semantics of the enclosed data. However, the semantics is only encoded in the name of the tag, and conflicts can be caused by homonyms and synonyms. Homonym means, the same name stands for different concepts in different sources. Synonym means, different names in different sources describe the same concept. Furthermore, a real world concept can be understood differently in different sources.

On data level, values may have the same semantics but use different representations. An example is the use of different units or currencies. The representation heterogeneity is caused by different conventions as well as erroneous data [SPD92].

Technical and data model heterogeneity can be solved by appropriate system architecture and software components. Schematic and partly semantic heterogeneity are subject of the schema matching and schema integration process. The schema integration process comprises homogenization of the local sources to provide an integrated, unified access to the data. Schema matching tries to find correspondences between different schemata. Schema mapping and integration resolve the integration conflicts using these correspondences. Overviews of schema integration are for instance given by Batini et al. [BLN86], Ram and Ramesh [RR99], and Conrad [Con97]. Rahm and Bernstein give an overview about schema matching principles and approaches [RB01].

Distribution

Information systems are characterized by types of data distribution. Central systems store data at a single site, whereas distributed information systems store and use data from different sites. Particularly, information systems for data integration utilize data from multiple heterogeneous, autonomous, and distributed sources. Hence, integration systems can be classified according to the data distribution into *materialized* and *virtual integration*.

Materialized integration. In the materialized case, data of the local data sources is copied to a central site. On the central site, the data is transformed into a global data model, the data is cleaned, and integration conflicts are removed. User queries are issued to this central database. Members of this class of data integration systems are Data Warehouse systems [Inm96] or Dataspace systems [FHM05, HFM06].

On the one hand, the materialization approach has several advantages. First of all, query processing is straightforward as all information is available about the data, its

structure, and its location. Furthermore, statistics and costs is used for query optimization and additional overhead introduced by network access is avoided. Another advantage is the independence of unreliable local sources. Since many local sources, e.g., stock and news tickers, update their data regularly and remove old data, historical data analysis is not possible. In contrast, materialization allows storing historical data as long as needed by the global system. On the other hand, materialized integration has also disadvantages. Data ages fast, thus, not the most current data is stored. Therefore, updates have to be made often, which cannot always be afforded. Furthermore, many data sources, especially in the WWW or Internet, do not allow the complete materialization of their data, because of data security or because of business and technical issues. These thoughts lead to the second class of data integration systems.

Virtual integration. In this scenario, data is kept in the local sources. Therefore, the global query system has to distribute queries to the local sources and to combine the local results to the required integrated result. Hence, the query planning and optimization is a complex process that is described later on in detail. The problems of the query optimization are missing statistics information, different query capabilities, and a large number of sources that are unreliable. Consequently, the query processing and the possibly slower query execution are disadvantages of the virtual integration approach. Representative kinds of this class of information systems are Federated database systems [SL90], Mediator systems [Wie93], Mashups [EBG+07], Meta-search [MYL02], or Deep Web search systems [CHZ05].

The distinction between materialized and virtual systems can be softened using partial materialization or caches on the global level, as discussed in [LC01, KSGH03, May05, CM08].

Classification of data integration systems

Based on the discussions above, information systems for data integration have to deal with autonomous, heterogeneous, and distributed data sources [SL90, Con97]. Another classification is proposed by Domenig and Dittrich [DD99]. It is based on the kind of materialization as well as the supported kind of queries. We complement this classification. It is illustrated in Figure 2.1. First of all, systems are classified into materialized and virtual integration. Data Warehouse systems [Inm96] are one example for materialized integration. Data Warehouses use a complete integration with an expensive ETL (Extract-Transformation-Loading) process. In contrast, Dataspace systems [FHM05, HFM06] use the pay-as-you-go semantic integration [SDK+07]. In this case, the data is materialized centrally but is not immediately integrated. In the first place, it coexists. Data integration will be carried out if it is necessary or during querying using special indexes [DH07]. Possible integration results can be materialized for later use. This approach allows a faster integration with less initial starting overhead than integration first approaches.

Systems using virtual data integration are further classified by the kind of queries they support [DD99]. If only unstructured queries are provided, we speak of Search and Meta-Search Engines [MYL02]. In the other case, structured queries over (semi-)

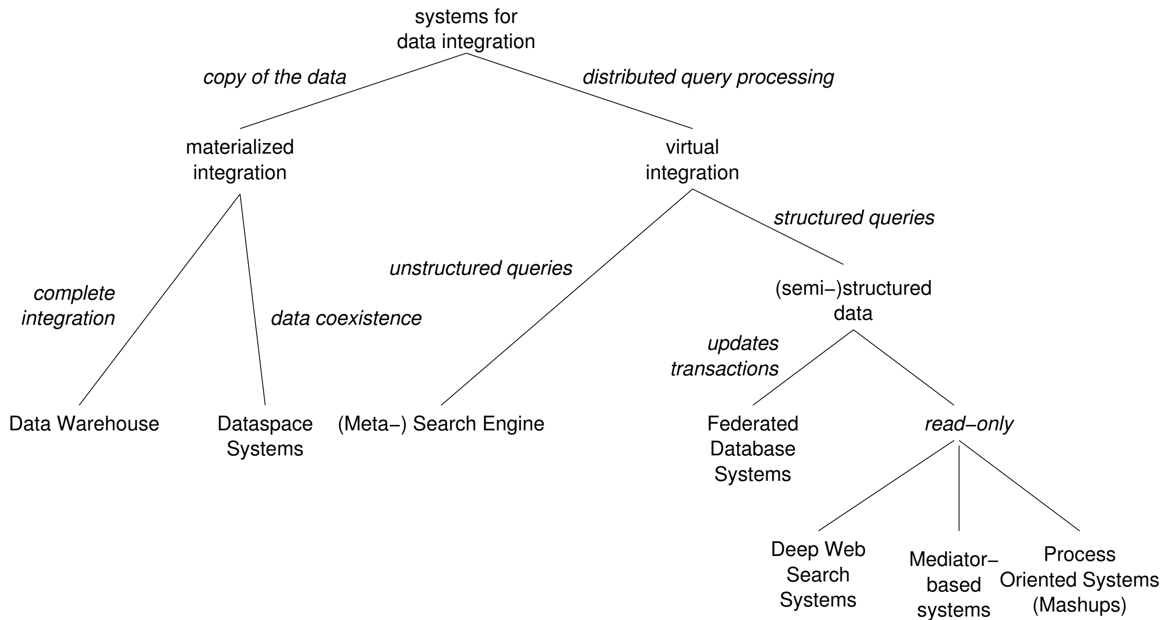


Figure 2.1.: Classification of systems for Data Integration based on [DD99]

structured data models are supported. Here, the systems are distinguished, whether they support only read-only queries, or they also support write operations. The latter systems require a tight "federation" and are mostly referred as *Federated Database Systems* as they try to support all DBMS requirements [SL90]. They are often designed by a bottom-up, (semi)-formal schema and data integration process. We distinguish read-only systems into *Mediator-based information systems* [Wie92], *Deep Web Search systems* [CHZ05], and *process oriented systems* (Mashups). Mashups integrate data from different Deep Web data sources or Web services by chaining inputs and outputs of different sources or by combining information from different services into one dataset. Example systems are Yahoo pipes[1] or MashMaker [EBG^{+}07]. We discuss meta-search engines and Deep web search systems in Chapter 4 in Section 4.4.1 and 4.4.2 in detail.

2.1.2. Mediator Systems

The term *mediator* in data integration was firstly introduced by Wiederhold [Wie92]. Global users and applications often require only a read-only access to the data and an integrated view to parts of the local data. Information sources in the Internet provide mostly limited query interfaces and additionally, one has to pay attention to the communication autonomy, because Web sources are changing often and can leave a federation at own will. Mediator-based information systems are designed to support that scenario. Wiederhold defined a mediator as follows.

[1] http://pipes.yahoo.com/pipes/ Last accessed: 2012-05-22

Definition 2.1 Mediator
*A **Mediator** is a software module that exploits encoded knowledge about some set or subsets of data to create information for a higher layer of applications [Wie92].* □

A mediator is a lightweight software component that mediates between the global user/application and the local sources. It provides an integrated access to the local sources for users or other mediators. Thus, the mediator is a service for other software components. Furthermore, mediator-based systems allow complete communication autonomy and support various kinds of information sources, i.e., structured, semi-structured, and unstructured data. Mediators support different integration methods and sources with limited query capabilities.

Mediator architecture

Supporting all techniques in one mediator would violate the demand on lightweight, easily manageable software components. Therefore, a mediator-based system usually consists of several, specialized mediators that are used by global applications and other mediators, as well. Consequently, the following three-tier architecture was proposed [Wie92].

The architecture of a mediator-based information system is illustrated in Figure 2.2. The system consists of three layers: *presentation layer*, *mediation layer*, and *foundation layer*. The main components are a set of *mediators* and *wrappers*. In Figure 2.2, lines denote queries against the global schema and integrated results, respectively; dashed arrows represent source queries in the global data model and not (or partially) integrated results. Dotted arrows stand for queries and the respective results in the local model.

Presentation Layer: Global applications and users utilize virtually integrated data provided by one or several mediators. The user sends global structured queries to an integrated schema. The results have to be integrated by the mediators and are presented following the global schemas. Global users have to know the mediators as well as knowledge about the global schema is required.

Mediation Layer: The mediation layer consists of several mediators and provides services for data integration. Each mediator can receive global queries of users and global applications. The queries are analyzed, and a query plan is created by the mediator. The query plan consists of the correct sequence of sources that have to be used to answer the query. Each mediator uses for this task *source descriptions* that provide information about the data stored in the sources and the query capabilities. Next, the plan is optimized and executed, that means, decomposed queries are sent to the sources. A source is either a wrapper to a source or another mediator.

The mediator combines and integrates the returned results and presents them in an integrated way to the global user and application. Integration conflicts caused by heterogeneity on different levels are reconciled. The mediators are developed by domain specialists and use metadata about covered sources and

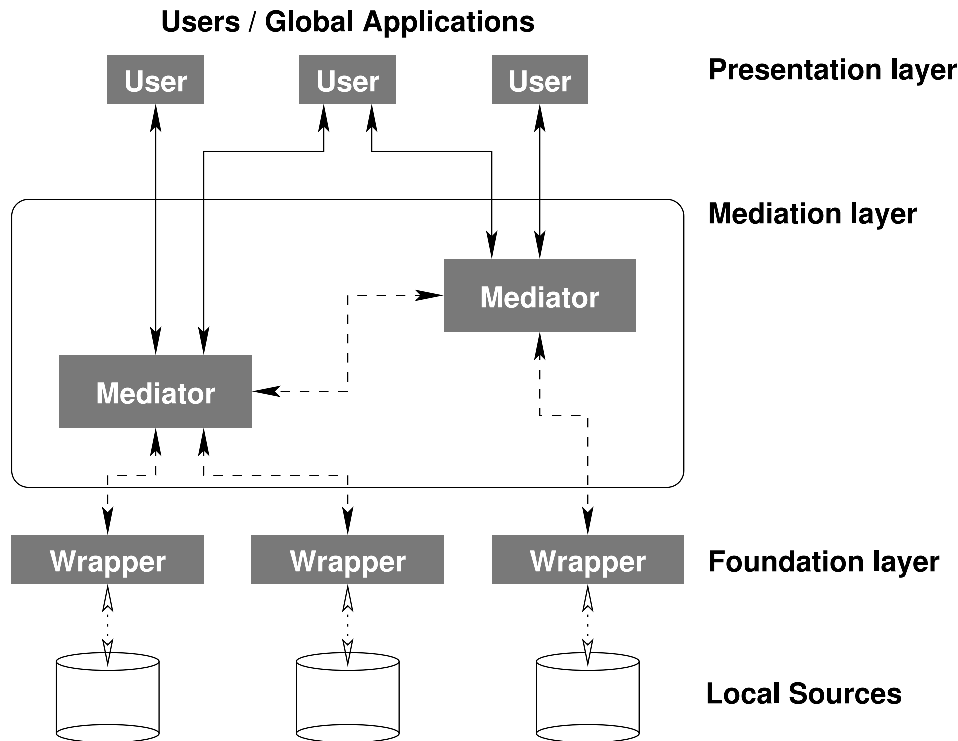


Figure 2.2.: Mediator architecture following [Wie92]

their data as well as their query capabilities. A domain model can be used to describe the global data.

The mediator stores all necessary metadata to fulfill the tasks: global schema, source descriptions, view definitions, reconciliation specification, etc.

Wrapper/Foundation Layer: Local sources in the foundation layer are accessed through *wrappers* and provide the data. Wrappers are software components that hide technical and data model heterogeneity. Wrappers provide information about the sources, e.g., information about the schema, query capabilities, cost information. The mediator extracts the information during the registration of a source at the mediator system and stores it in the global metadata catalog. The tasks of wrappers include the translation of source queries in a way, that local sources can support them, emulation of not supported operations, and translation of the results into the global data model or mediator internal data processing data models. The wrapper contains technical metadata that provides information about data model translations between local source and mediator data model.

Mediator query processing

An important task of the mediator is to execute global queries that were issued against the global schema. Figure 2.3 gives an overview about the query processing [BKLW99].

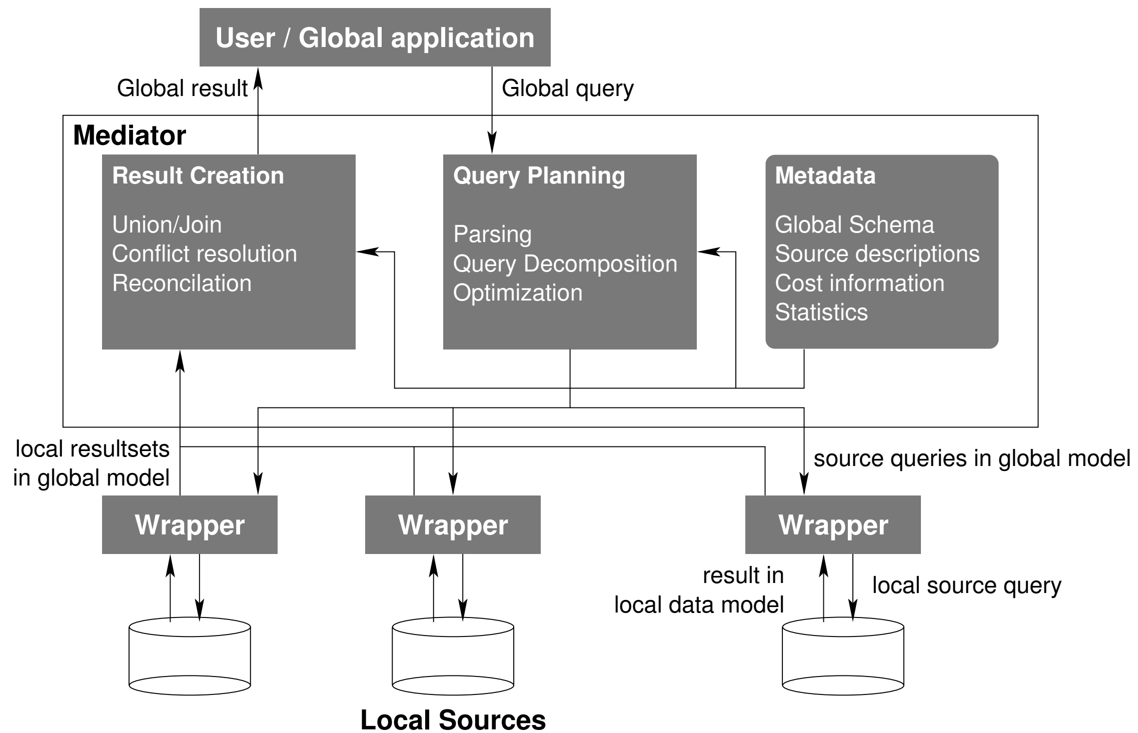


Figure 2.3.: Mediator query processing based on [BKLW99]

A user or a global application sends a global query against the global schema. The mediator receives the query and parses the query in the first step. The next step has the goal to create a plan of source queries that are necessary to compute the query result. The plan is created using *source descriptions* that describe correspondences of the content of the local source according the global schema [SPD92]. The source descriptions are stored in the metadata catalog of the mediator. There are two main approaches of source descriptions that have influence to the kind of query planning: *Global-as-View (GaV)* and *Local-as-View (LaV)* [Len02].

Global-as-View (GaV): Following the GaV approach, each global concept is defined as one or more views over the local sources. Hence, each correspondence rule defines the semantics of a global concept with the help of source queries. For query translation, it means that the global views are expanded to the corresponding source queries. The query decomposition step is more or less coded in the view definition. The disadvantage of the GaV approach is that changes in one local source invalidate view definitions of all global classes that use the source. The view definitions have to be recreated.

Local-as-View (LaV): The LaV approach tries to mitigate the disadvantage of the GaV source description in order to support mediation of frequently changing environments of Web sources. Local-as-View means, that local classes or concepts are expressed as views over the global schema. Each local source is understood as a part or view of the global data space. During query processing, every view may contribute to the answer of a global query. It leads to the problem of answering queries

using only views [Hal01]. The system has to find a plan that uses only views instead of the global relation. At first, the problem was introduced for the usage of materialized views [LMSS95]. However, in data integration scenarios the number of views is large and the goal is to find a maximal-contained query instead of an equivalent query [Ull97, Hal01]. Proposed algorithms are among others the bucket algorithm [LRO96], the inverse-rule algorithm [DG97], and MiniCon algorithm [PH01].

A combination of both LaV and GaV was proposed in Friedman et al. [FLM99] to support data webs that are typical in Web environments. The authors propose the so called GLaV (Global-Local-as-View), where on both sides of the correspondence rules are views.

After finding the correct source query plan, either using LaV or GaV rewriting, the system tries to find the best execution plan of the rewritten query. Mostly the optimization tries to find the plan with lowest costs [HKWY97, TRV98, LRO96]. Other approaches generalize that notion and try to find the plans with the best quality, which is a critical issue in data integration [NLF99].

The optimization step relies on information about query capabilities of the sources and the corresponding costs provided by the wrapper during source registration. After the final plan is constructed, the plan is executed. First, the source queries are sent to the wrappers. The wrappers translate the queries such that, that the local API can handle the request. If a source cannot support some query operators, these operators are executed in the mediator or the wrappers.

The returned results, expressed in the local data model, are translated by the wrapper into the global data model or the data processing model. Subsequently, the results are sent to the mediator again. The next task of the mediator is the integration of the source results. Here, different approaches exist [BKLW99]. *Collection* expresses, that the mediator simply collects the objects from different sources. A mediator can try to use *object fusion* to fuse semantically equivalent objects of different source into one object. Semantically equivalent objects refer to the same real world concept [Ken91]. Object fusion has to deal with data level conflicts, i.e., semantically equivalent values are represented differently. These conflicts have to be solved by specific joins, reconciliation functions, similarity functions, or mapping tables. Further approaches are *abstraction* to overcome semantic heterogeneity and *supplementation* to use unstructured data with the help of metadata.

Finally, the integrated result is presented to the user. Possibly, information about quality, sources, and other metadata are presented to the global application and user. That may be beneficial to increase the understandability of results.

2.2. Concept-based Mediator Systems

Nowadays, the World Wide Web (WWW) and large company intranets provide a large number of information sources. Because of heterogeneity and a high degree of autonomy of the sources, solely structural integration is hard to handle. A similar problem exists in the WWW, where the data is provided in the form of HTML files. In order to manage this vast amount of data, Berners-Lee proposed the idea of the Semantic Web [Sem05]. In this vision, the semantics of the data in Web pages is

described explicitly by using different domain specific ontologies expressed in languages like RDFS [BG03] or OWL [BvHH+03].

Similar problems arise in the context of mediator systems integrating a large set of WWW sources. Integration only based on structured integration suffers of increased autonomy, heterogeneity, and distribution [She99]. In this context, Sheth [She99] classifies the systems providing interoperability into three generations. The first generation tried to overcome system-specific and structural heterogeneity. The number of participating sources is relatively small, and systems are mostly relational or object-oriented database systems. The second generation deals with the read-only integration of a large number of sources. The emphasis of these systems is on reconciliation of syntactical and schematic conflicts. Different kinds of sources are integrated: structured, semi-structured, and unstructured sources. Top-down integration paradigms are used to deal with all kinds of autonomy. Finally, the third generation of data integration systems deals with the semantics of the data and user queries to provide interoperability. Halevy et al. made a similar observation [HRO06].

2.2.1. Principles

The borders between the generations are not sharp, but the trend is from the system and syntactical problems to semantic problems. *Concept-based* mediator systems are one kind of systems of the third generation. In general, a concept-based mediator uses an ontology [Gru91] to describe the concepts and properties of the data stored in various information sources. As a "world" ontology is not possible, the ontology of a concept-based mediator is domain-specific and is constructed by domain experts. Users issue concept-based queries to the global concept model to obtain results. The connection between the Artificial Intelligence and database communities is discussed by Noy et al. [Noy04] and Doan et al. [DH05]. Wache et al. describe the characteristics of semantic integration systems in general [WVV+01]. We will focus here on concept-based mediator systems. The mediator-wrapper-architecture provides in this context all technical support to mediate between a concept-based (semantic) query and data that resides in the local sources. Summarizing, concept-based mediator systems solve two problems:

1. If data is related, but information comes from sources of different worlds, then concept models allow the definition of connections between the sources in a homogeneous manner [LGM01].

2. The structural heterogeneity between local sources is high because structured, semi-structured, unstructured sources, or totally different representation make a purely structural integration not possible, but the integration of the semantics is enabled by concept-based mediators.

In the following, we investigate concept-based mediator systems based on the aspects *(i)* ontology as global schema, *(ii)* kinds of source description, *(iii)* concept-based query language, and *(iv)* query processing and optimization.

Ontology as global schema

The first step of concept-based integration is the top-down definition of the global, domain specific ontology. The ontology defines the vocabulary and the semantics of the data in the respective area. The ontology is the way the user uses the integrated data. The concept schema describes the intentional part of an ontology, while the sources provide the instances.

Different data models are used to define a concept-based schema. Exemplary data models are the general concept model (GCM) [LGM01], description logic [LRO96], contexts [GBMS99], or proprietary models [SH07]. Furthermore, Semantic Web languages like OWL and RDFS are used [ABFS02a, LF04, SGS05, Lan08, CGL^{+}10].

A further use of the ontology is the validity test of the integrated data, e.g., testing whether the integrated data objects satisfy the global ontology constraints [GBMS99, LGM01].

Source descriptions

To answer semantic queries, the mediator uses source descriptions that map the local data to a certain global concept. That means, the source descriptions explicitly model the semantics of source queries. In that way, the heterogeneity of the sources is solved on schema level as well as on data level.

There are two kinds of mapping in semantic integration using ontologies. First, there are mappings between ontology sources. Second, there exist mappings between a global ontology and local sources. We focus here on the latter as part of a concept-based mediator system. We denote the mappings as source descriptions. We classify source descriptions according to two criteria: what is described and how it is described.

The first criterion describes what is mapped to the global concept-schema. On the one hand, sources describe their data in a concept-based model. Sources can use the given global ontology. Here, the data is exported as part of the global ontology [Lan08, LWB08]. Another approach assumes, the local data is expressed in a local ontology. Here, local concepts are added to the global ontology using global concepts as anchors [LGM01]. On the other hand, global ontology concepts are mapped to local data types [ABFS02a, SGS05, CGL^{+}10, LF04, BGTC09]. The local data types are expressed in any data model, for example, as XML or relational data.

The second criterion (how are descriptions described) comprises the kind of source descriptions which are GaV, LaV, or GLaV. An example of the GaV is the KIND system [LGM01]. The LaV approach is used in [ABFS02a, SGS05], for example, while Calvanese et al. [CGL^{+}10] describe a GLaV approach. The mappings can map to concepts and data properties as well as to derived concepts [ABFS02a], allowing fine-grained query rewriting. In order to provide global relationships, two approaches exist: first, creation of global keys from the attribute values [ABFS02a] or second, the definition of join mappings for object properties [SGS05]. Both are expressed using the global concept model.

Concept-based query languages

This classification dimension describes the query language. Used query languages are classic ontology languages like F-Logic [KLW95] and Description Logic combined with datalog [LRO96]. Furthermore, systems use restricted semantic web languages like subsets of SPARQL [Lan08, LWB08] or OWL QL [CGL+10, LF04]. Other systems use adapted OQL-like languages [ABFS02b] or derivations of XQuery [SGS05] or XPath [SH07]. The provided operations have to include concept level operations like traversal of concept hierarchies and other relationships, concept-selection, semantic reasoning as well as operations on the data level, e.g., selection or projection.

Query processing and optimization

The query planning in different systems depends on the source descriptions. GaV-based systems resolve the global views and create the union of the source query results, e.g., [Lan08, LGM01]. LaV-based systems like [ABFS02b] use and adapt known algorithms as the MiniCon approach [PH01]. The YACOB system [SGS05] combines the extensions of concepts from different sources first and applies global joins afterwards. Thereby, all connections between concepts are expressed as global joins. Calvanese et al. showed the GLaV rewriting of OWL-based integration systems [CGL+10].

2.2.2. Approaches and Systems

In this section, we briefly discuss several systems. Well-known systems like Garlic [CHS+95], DISCO [TRV98], TSIMMIS [GMPQ+97], its successor MIX [BGL+99], Information Manifold [LRO96], and HERMES were often discussed in literature [She99, BKLW99]. Instead, we focus on concept-based mediators.

Information Manifold (IM) [LRO96] is similar to concept-based mediator systems as it uses a world view modeled by an extended relational model as global view. Local sources map their schema elements to the world view using a LaV approach. However, IM targets mainly to the reconciliation of heterogeneity on the structural level and does not explicitly model a global ontology or vocabulary.

SIMS [AK93, AKS96, AHK97] The SIMS mediator system is used in a single application domain. It uses the concept-based language LOOM to model the application domain. LOOM allows the modeling of classes, their relationships, and their roles. An information source is also modeled in terms of the global model. The local concepts are mapped to the global concepts using the LaV approach, i.e., local concepts and relations are mapped to the global concepts and relations, respectively. Global queries are expressed by means of the LOOM model. The SIMS system supports structured data sources.

SIMS's successor **ARIADNE** [AAB+98, KMA+01] extends the mediator system for semi-structured data provided by Web sources. ARIADNE uses the same LOOM model for description of the application domain. The system extends SIMS by providing features of modeling Web page data with LOOM. Furthermore, the query planner

of SIMS is extended to deal with a large number of sources, especially Web sources that are characterized by limited query capabilities.

Context Interchange (COIN) [GBMS99] Similar to the other systems, the Context Interchange system uses a domain specific model for description of the application domain. The COIN model comprises primitive types and primitive objects as well as semantic types and objects. Primitive types correspond to data types that are native to the sources. Semantic types are complex types that support the data integration. The collection of both kinds of types provides the common type system for integration. Elevation axioms provide the mapping to the global domain model. The crucial part is the set of context axioms that describe the data in a given context, i.e., one data value has different meanings in different contexts, e.g., sources. Thus, the source description follows the LaV approach. The mediator uses all this information to mediate during runtime attribute domain conflicts, e.g., it can use modification functions or even find other sources needed for conversion.

InfoSleuth [JBB+97] InfoSleuth is an agent-based semantic integration system. It extends the system Carnot [SCH+97] to support new challenges induced by WWW information sources, e.g., increased autonomy in design and communication as well as their large number. InfoSleuth uses an agent-based architecture, i.e., the tasks of the three layer mediator architecture are distributed over a number of specialized agents that communicate using KQML. Besides infrastructure agents as Broker and Monitor agents, a number of software components implement the mediation services. User agents present the common interface of the system to the user. That means, they are the presentation layer. Users issue queries against domain models (ontologies), which are managed by domain agents. This behavior is similar to concept-based mediators. In the next step, the query is sent to task planning and execution agents which use information of resource agents to select a correct execution plan. While the task planning and execution agents represent the query planning and execution of a mediator, resource agents provide the source descriptions and wrapper functionality. Putting all together, InfoSleuth provides an extensible and distributed concept-based mediator system.

KIND [LGM01] Ludäscher et al. proposed the concept-based mediator system KIND to deal with the "distinct world" problem. In this case, two sources do not overlap on the structural level, even do not show structural conflicts, but the data can be interconnected on semantic level using additional domain knowledge. KIND uses domain maps as high-level description of the domain. The domain map serves as a navigational map in the application domain. Local sources and their data are already lifted to the conceptual level by wrappers, where each source can be expressed in a local conceptual model. Translators are used to transform the local conceptual model into the general concept model (GCM) that is used by the mediator. KIND uses F-Logic [KLW95] as GCM. The wrappers provide specific concepts that describe where the local source is located in the domain map, i.e., the concepts of the domain map are semantic anchors of the sources. Integrated views are defined using domain maps

with added conceptual source data. The view definition follows the GaV approach. In that way, structurally not connected data is connected and is queried in an integrated way.

STYX [ABFS02a, FAB+02] Styx is a concept-based mediator that is designed to integrate XML data sources, i.e., all sources export their data as XML data. A lightweight concept model is used to describe the application model. The concept model describes structured objects and relationships. Data values are not considered. Source descriptions map the local data directly to the global domain model. The system utilizes the LaV approach to model local paths by means of paths in the concept model. Since the LaV source descriptions are used, the query processing is designed following the MiniCon approach [PH01], which creates a query plan by considering the subgoals of a conjunctive global query as well as the join conditions.

SemWIQ[Lan08, LWB08] The SemWIQ system uses an RDF Schema (RDFS)-based global ontology. Each source exports its data in RDF using the global types. Thereby, wrappers provide this functionality for non-RDF sources. Each source supports a SPARQL subset using this approach. The mediator user expresses its query in a SPARQL subset. The system transfers the query first in a global algebra representation. Afterwards, the SemWIQ mediator uses source descriptions to replace global concepts referenced in the query with union of local queries. The focus of the system lies in the optimization of the queries and the efficient processing. SemWIQ supports all sources that provide RDF data and allow SPARQL queries. These functionalities can be provided through RDF wrappers to many kinds of data sources like relational databases and text files.

2.2.3. Other approaches

The previous sections showed the usage of an ontology for concept-based mediator systems. An ontology can also be used to provide interoperability between heterogeneous data sources in other ways. Doan and Halevy focus on ontology matching and mapping of sources to the ontology [DH05]. They provide an overview of challenges of ontology management for data and schema integration. The work of Noy is the counterpart paper from the Artificial Intelligence point of view [Noy04]. She examines the problems of ontology integration (merging) and the usage of ontology mappings. An ontology can also be used for query expansion of a keyword or XPath queries over heterogeneous sources [TW02a]. Calvanese et al. studied OWL as integration language [CGL+10]. They investigated the complexities of OWL queries in the light of different mappings (LaV, GaV, GLaV). Lehti and Fankhauser used an ontology, expressed in OWL, to integrate heterogeneous XML data sources [LF04]. They directly map the XML Schema expressions to global concepts and properties. Hornung and May describe how to query Deep web sources with semantic annotations and SPARQL [HM09b]. Deep Web sources are modeled as n-ary predicates with input and output variables called signatures [HM09a]. Subsequently, these predicates

are annotated in three levels: technical, signature, and semantic level. The source descriptions are modeled local-as-view.

2.3. Summary

In the previous sections, we described the characteristics of information systems providing interoperability between autonomous, heterogeneous, and distributed information systems. Subsequently, the architecture and issues of mediator-based information system were discussed. Mediator systems are characterized by the mediator-wrapper-architecture, the typical top-down integration and read-only access to the sources. Hence, they provide a support for often-changing, highly autonomous sources with limited query capabilities, e.g., WWW sources or Intranets of large companies.

Dealing with highly autonomous, heterogeneous WWW sources using only structural integration techniques led to problems. Thus, concept-based mediator systems have been emerged for the recent years. Concept-based mediators use, as global integration model, a semantic description of the data in the form of application domain ontologies. This is similar to the Semantic Web where semantic descriptions are used to make heterogeneous content manageable and connectable. Concept-based mediators use domain models as navigation help for users as well as a solution of the distinct world problem, i.e., finding relationships of data that is only given on the semantic level, and not on the structural level. Furthermore, data conflicts can be solved using semantic descriptions of the context of the data [GBMS99]. In summary, concept-based mediator systems provide a comprehensive framework for flexible integration of information sources.

Concept-based mediator systems provide powerful query languages that allow complex queries over the application model. The usage of these queries as presentation layer is too complex for a normal user, because it requires a deep knowledge of the query language as well as the application domain. A possible solution is the use of canned queries. Canned queries are predefined queries for a specified task that can be parameterized by the user. This kind of queries is presented by a query form. However, for data integration these queries are not flexible and powerful enough.

Another possibility of a presentation layer is a visual interface that provides two functionalities: browsing of the stored data like the interface BBQ of the mediator systems TSIMMIS and Mix [MP00] and graphical query languages like QbE [Zlo75]. In a concept-based mediator system, browsing can be done along concepts and their relationships. For example, it is possible to go down a specialization hierarchy. Furthermore, the browsing approach can be supplemented by canned queries. That means, the properties of the current concept are presented to the user, and she/he can fill in some values for the properties which represents a simple selection. However, that approach does not allow complex queries to get new relationships. They are not powerful enough. Visual query languages help to understand the structure of the data as well as provide support to formulate complex queries, but they tend to be too complex for normal users.

Because of these problems, Davulcu et al. [DFKR99] proposed the structured universal relation as the presentation layer in their Web integration approach. The structured

universal relation is an extension of the universal relation [Ull90]. A universal relation offers a list of all attributes in the database, and computes automatically the necessary joins in the normalized relation database to answer queries against the universal relation. The structured universal relation relaxes some requirements and groups the attributes based on a concept hierarchy. The attribute grouping is already given in a concept-based mediator.

An extension of this approach is the combination of *keyword search* and *concept-based search*. A keyword query consists only of a set of keywords as well as Boolean operations between them. Concept-based queries are formulated by means of the global concept-model and a concept-based query language like CQuery [SGS05].

The keyword search over concept-based mediators has the following advantages:

- Simple query interface: the user has to formulate the query simply by some keywords. This kind of interface is common in the WWW and it is known from famous search engines, e.g., Google, Bing, etc.
- Use of integrated data: the data is integrated by the mediator, i.e., heterogeneity is removed, and many sources are queried in a uniform fashion. Furthermore, results are integrated, and data conflicts are removed.
- Use of the domain model: the domain model can be used as semantic index of the data. Furthermore, similar terms are closely connected in the model. Thus, we can add semantic descriptions for query expansion. For instance, if a user searches for paintings, the system can also propose instances of graphics, which is a similar concept.
- Integration of structured search: the combination of structured search and keyword search allows the user to formulate complex queries in a simple way. By structured queries, the search space is specified, e.g., by selecting concepts. Keyword search relaxes the problem of complete knowledge of the model.

This kind of keyword search resembles to keyword search over structured and semi-structured databases [YQC10, PgL11]. The integration of keyword search into a concept-based mediator system imposes new challenges. These are:

- **Integration of keyword search into the concept-based data model:** Information is spread over different concepts and attribute. We have to define information units suitable for keyword search, e.g., a network of objects and concepts.
- **Mapping of keyword queries to global queries as well as to source queries:** The data is virtually integrated. We have to create global and local queries to obtain the data. First, we have to map keywords to global single concept queries. We have to define which information is needed for this task. Second, we have to combine concept queries to queries that connect different concepts and their instances.
- **Efficient execution:** Many queries can be generated by keyword queries. We have to find efficient execution approaches to reduce the number of queries to the sources and the number of transferred objects.

These points have to be solved to integrate keyword search successfully into concept-based mediator systems with the goal of efficient and effective user interfaces. In the following chapter, we introduce the YACOB mediator system. The system is a concept-based mediator system and provides the data integration for the proposed keyword search system.

3. Yacob – Yet Another Concept-Based Mediator System

After giving an overview of principles and systems in Chapter 2, we now describe the concept-based mediator YACOB. As Figure 3.1 illustrates, YACOB is the integration component of the proposed keyword search system. The mediator provides concept-based access, integration, and query planning. The YACOB system was proposed by Sattler et al. [SGHS03]. This chapter is based on Sattler et al. [SGHS03, SGS05] and shares material with these studies. The YACOB system uses a concept-based schema and a local-as-view mapping scheme to integrate XML data sources. In particular, YACOB supports Web data sources. In the remainder of the chapter, we describe the integration model (Section 3.1), the used query language CQuery and the corresponding algebra (Section 3.2), and the query planning and processing in YACOB (Section 3.3). We conclude the chapter with a summary (Section 3.4).

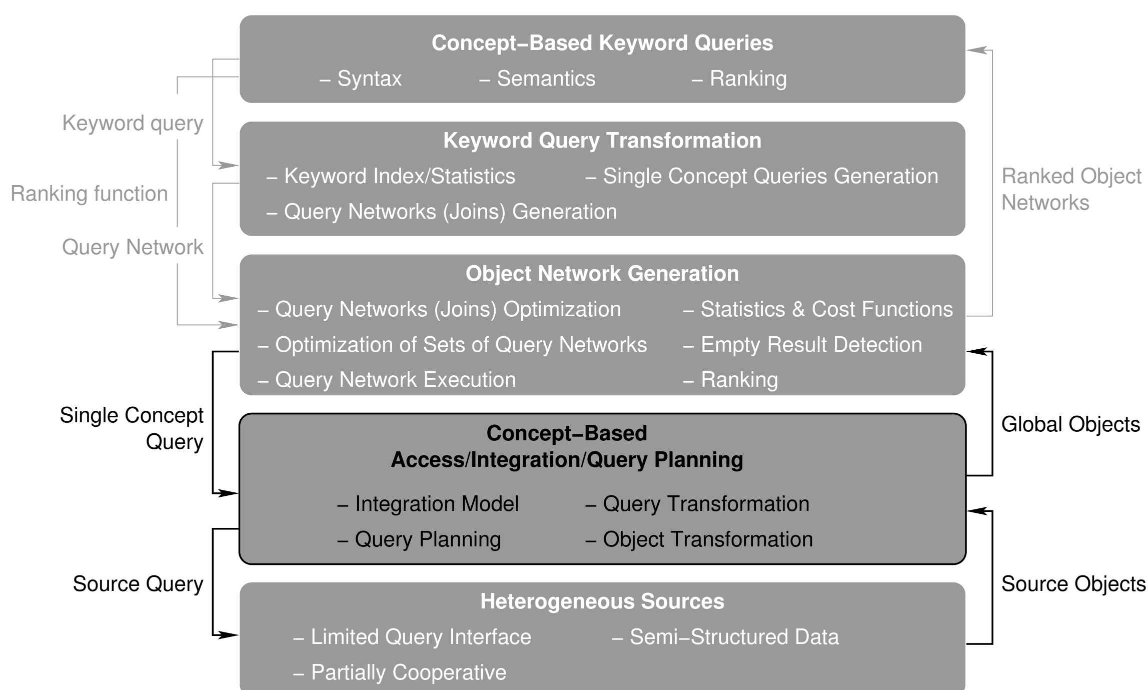


Figure 3.1.: Overview: Integration

3.1. Integration Model

The integration model of YACOB consists of a *concept model* that allows modeling of the global application domain and a semi-structured *data model* that represents the actual data values. The data model is mainly used for query processing and data exchange. Finally, the source descriptions in the form of LaV mappings are the third part of the integration model. While the data model is XML-based, the concept model and the mappings extend RDF Schema (RDFS) [BG03] and are implemented in RDF.

3.1.1. Concept model and Data model

The YACOB concept model is designed to represent the semantics of structured objects but also the semantics of data values, e.g., of categorical data. Therefore, we extend the RDFS model by introducing the constructs *concept* and *category*. Both constructs are subclasses of `rdfs:Class`.

A concept is a class that has an extension consisting of objects. Objects are built of data from different local sources. Properties are attached to concepts. They describe the features of objects. A property models either a relationship between two concepts (denoted as concept property), between a concept and a literal (literal property), or between a concept and a category (categorical property).

A category is a class that does not have an extension. Instead, categories describe categorical data values. The set of categories is disjoint from the set of concepts. Categories allow the definition of vocabularies that provide a uniform representation of categorical data values, which might be differently encoded in different sources. Equivalent to concepts, categories are organized in hierarchies but are only allowed as range of properties.

Figure 3.2 illustrates an exemplary concept schema modeled using concepts, categories, and properties. Firstly, two concept hierarchies exist in the schema. They model the semantics of `culturalAssets` and `painter`[1]. The arrows represent the `rdfs:subClassOf` property defined in RDFS. Furthermore, there are properties that describe the relationships between two concepts, e.g., `paintedBy`. The property `paintedBy` explains that a painting (or a sub-concept) is painted by a painter (or sub-concept), i.e., the domain ranges over the objects of concept `painting`, and the range is the class `painter`. The categorical values of a property are described by categories as seen for property `portrays`. The category hierarchy `motif` describes explicitly different kinds of subjects that are normally hidden in the sources and represented as strings using different encoding schemes. Therefore, the user has to know only the global values. In summary, the YACOB concept model relies on RDFS and extends it by the constructs concept and category. In the following, we define the model formally.

Equivalently to RDFS, all schema constructs are identified by a Uniform Resource Identifier (URI) and a valid name. Values are always of the literal type as defined in [BG03].

[1] The concept `painter` is exemplary for artist concepts.

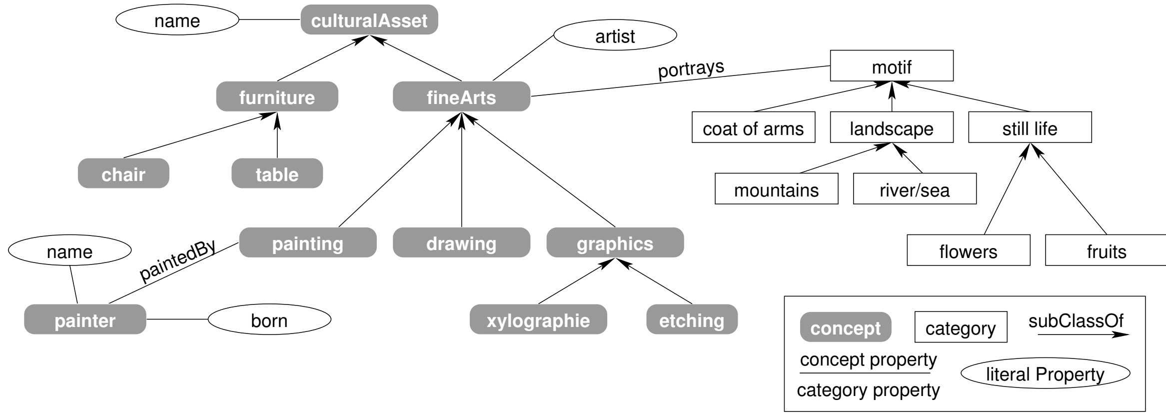


Figure 3.2.: Exemplary YACOB concept schema

Definition 3.1 URI, Name, Literals
The set of Uniform Resource Identifiers (URI) is denoted by URI. The set Name comprises all valid names, and $\mathcal{L}$ denotes the set of literals. □

Using the sets *URI*, *Name*, and $\mathcal{L}$ we define the YACOB concept model as follows.

Definition 3.2 Concept model
*The **concept model** consists of the parts:*

- *the set of **classes** is defined as $\mathcal{K} \subseteq URI \times Name$, i.e., all classes are of the form $(uri, name)$,*
- ***concepts** ($\mathcal{C} \subset \mathcal{K}$) are defined as classes that have object extensions in the sources,*
- ***categories** are classes, i.e., $\mathcal{V} \subset \mathcal{K}, \mathcal{V} \cap \mathcal{C} = \emptyset$, that represent abstract property values and do not have object extensions,*
- ***properties** are assigned to classes, and the set of all properties is defined as $\mathcal{P} = Name \times \mathcal{C} \times \{\mathcal{K} \cup \{\mathcal{L}\}\}$, given a property $p = (name, c, v)$ we say either*
  1. *p is a concept property, if $v \in \mathcal{K}$,*
  2. *p is a category property, if $v \in \mathcal{V}$, or*
  3. *p is a literal property, if $v = \mathcal{L}$,*
- ***specialization relationship is_a** $\subset \mathcal{K} \times \mathcal{K}$, i.e., if $(c_1, c_2) \in$ **is_a**, c_1 is a subclass of c_2 and c_2 is the super-class of c_1. In addition, the hierarchies of concepts and categories are disjoint: if $(c_1, c_2) \in$ **is_a** then either $c_1, c_2 \in \mathcal{C}$ or $c_1, c_2 \in \mathcal{V}$. Properties are inherited by sub-concepts: it holds for two concepts $c_1, c_2 \in \mathcal{C}$: if $(c_2, c_1) \in$ **is_a** (c_2 is derived from c_1), then $\forall (p, c_1, v) \in \mathcal{P} : (p, c_2, v) \in \mathcal{P}$.*

□

The concept model allows the construction of a global concept schema that consists of a set of concepts and categories, which are instances of the types concept and category and are organized in hierarchies. Furthermore, the schema contains properties defined to the given concepts and categories. The schema is defined as follows.

Definition 3.3 Concept schema
The ***concept schema*** *is a 4-tuple* $S = (\mathbf{C}, \mathbf{V}, is_a, \mathbf{P})$ *consisting of a set of concepts* $\mathbf{C} \subseteq \mathcal{C}$*, a set of categories* $\mathbf{V} \subseteq \mathcal{V}$*, a set of properties* $\mathbf{P} \subseteq \mathcal{P}$ *assigned to the concepts in* $\mathbf{C}$*, and a set of* ***is_a*** *relationships.* $\square$

After we have described the concept level, we define the data model as well as the transition between data and concept level. Yacob uses a semi-structured data model to represent instances of concepts. These elements of concept extensions are denoted as objects. The data model is mainly used for query processing, data exchange, and queries between sources and mediators. The global query formulation is entirely based on the concept model.

The data model is defined similarly to the OEM model used by the mediator system TSIMMIS [PGMW95]. Objects are triples that comprise a unique object identifier, an element name that describes the object as well as an object value. The value can be either an atomic value, i.e., a literal, an object identifier, i.e., a reference to another object, or a set of object references. In this way, semi-structured objects like nested XML elements are supported. The following definitions formalize the data model of Yacob.

Definition 3.4 Object Identifiers
The set of all object identifiers is denoted as $\mathcal{I}$*. The power set of* $\mathcal{I}$ *is* $\mathbb{P}\mathcal{I}$*.* $\square$

Definition 3.5 Data model
The ***data model*** *is defined as follows: Let* $\mathcal{O} = \mathcal{I} \times Name \times \{\mathcal{L} \cup \mathcal{I} \cup \mathbb{P}\mathcal{I}\}$*, where* $(id, name, val) \in \mathcal{O}$ *consists of a unique object identifier id, an element name (name), and a value val. The value val represents either an atomic value (literal,* $val \in \mathcal{L}$*), an object identifier (representing an object reference* $val \in \mathcal{I}$*), or a set of object identifier* $val \in \mathbb{P}\mathcal{I}$*.* $\square$

The extension of a concept is a set of objects. The element name of each object equals to the concept name. Such an object is the root of an object tree and has as value object references. Every referenced object corresponds to a property defined for the concept. The extension of a concept is defined as follows.

Definition 3.6 Concept extension
The ***extension*** $\boldsymbol{ext} : \mathcal{C} \rightarrow \mathbb{P}\mathcal{O}$ *of a concept* $c = (uri, name)$ *comprises a set of instances with an element name equal to the concept name and a set of identifiers val referring to the properties defined for the concept:*

$$\begin{aligned}\boldsymbol{ext}((uri, name)) = \{o = (id, elem, val) | elem = c.name \wedge \\ \forall i \in val \exists c', val', pname : (i, pname, val') \in \mathcal{O} \wedge \\ (pname, c, c') \in \mathcal{P} \wedge val' \in c'\}.\end{aligned}$$

$\square$

3.1.2. Source mappings

The second part of the integration model is the mapping of local schemata to the global concept schema. That is, the description how a source supports the global concepts and how the structure of local objects fits to the properties of the global concepts. The mapping model of YACOB follows two main principles:

1. the mapping model uses the RDFS paradigm, which means that concepts and properties are mapped independently, and

2. the mapping model follows the local-as-view approach.

A mapping consists of a local description and a global concept schema element. The former is denoted as the left-hand side (lhs) and the latter is denoted as the right-hand side (rhs) of the mapping. The mapping is then $lhs \rightarrow rhs$. RDF classes implement the structure of local descriptions. The descriptions are, therefore, instances of the classes. The mapping for concept schema elements is implemented by an RDFS property `providedBy`. In the YACOB system, we distinguish between concept, property, and value mappings for global concepts, properties, and categories, respectively. Furthermore, the YACOB mappings comprise join mappings that describe concept properties as value joins between global concept extensions.

Concept mappings

An instance of the *concept mapping* class describes how a source provides objects to an extension of a concept. Thereby, there are two cases: on the one hand, a source contains objects that contain information for all global properties, on the other hand, sources provide only partial objects according to the given global schema. We define concept mappings as follows.

Definition 3.7 Concept mapping
*A **concept mapping** is defined as*

$$cm \rightarrow c$$

with cm a 3-tuple $(Source, LName, FilterPredicate)$, where Source is the name of the local source, LName the local element name representing instances of the concept, and FilterPredicate is an XPath expression allowing a further specification of the instance set. We denote the set of all concept mappings as $\mathcal{M}^{\mathcal{C}}$. The set of mappings assigned to one global concept c is denoted as $CM(c)$, i.e., $CM(c) = \{(cm \rightarrow c) | (cm \rightarrow c) \in \mathcal{M}^{\mathcal{C}}\}$.
□

During query processing, the local source name *Source* allows the identification of the source. The local element name denotes the XML elements used in the source to represent instances of the global concept. The *FilterPredicate* is an XPath expression. Filter predicates allow the selection of objects based on distinguishable local properties. A concept mapping instance is attached to a global concept.

Property mappings

The second part of the mapping model consists of *property mappings*. A property mapping describes how a local element *LName* represents a global property. The property mapping consists of a source name and an XPath expression and is attached to a global property.

Definition 3.8 Property mapping
*A **property mapping** is defined as $pm \to p$ with*

$$pm = (Source, PathToElement),$$

where Source is the name of the local source and PathToElement represents an XPath expression to the local XML element representing the property. The set of all property mappings is denoted as $\mathcal{M}^{\mathcal{P}}$. The set PM(p) contains all property mappings assigned to property p, i.e., $PM(p) = \{(pm \to p)|(pm \to p) \in \mathcal{M}^{\mathcal{P}}\}$ □

Join mappings

We map concept properties to join operations between the extensions of the connected concepts. The join explains the semantics of the connection. We denote this kind of mapping as *join mapping*. A join mapping is expressed by means of the global model. That is, the join is expressed using global concepts and properties.

Definition 3.9 Join mapping
*A **join mapping** is defined as $jm \to p$ with*

$$jm = (SrcProp, TgtConcept, TgtProp, JoinOp),$$

where SrcProp describes the source property, TgtConcept and TgtProp denote the target's concept and property, respectively. JoinOp represents the kind of the join operation, e.g., equi- or similarity join. The set $\mathcal{M}^{\mathcal{J}}$ comprises all join mappings, and JM(p) is the set of join mappings to a property p, i.e., $JM(p) = \{(jm \to p)|(jm \to p) \in \mathcal{M}^{\mathcal{J}}\}$
□

For example, the property `paintedBy` illustrated in Figure 3.2 is expressed as join between the concepts `painter` and `painting` with the join condition `artist=name`. That means, the corresponding join mapping instance is the tuple $(artist, painter, name, =)$.

Value mappings

Categories describe in the concept model local categorical values in a uniform way. It is assumed that categories are represented as literals in the sources in different ways. A value mapping instance is a pair of a source name and a literal. It is attached to a category.

Definition 3.10 Value mapping
*A **value mapping** is of the form $vm \to v$ with $v \in \mathcal{V}$ and*

$$vm = (Source, Literal)\,,$$

where Source denotes the source name and Literal denotes the local value that represents the category. $\mathcal{M}^{\mathcal{V}}$ *describes all value mappings, and* $VM(v)$ *is the set of all value mappings assigned to category* v, *i.e.,* $VM(v) = \{(vm \rightarrow v)|(vm \rightarrow v) \in \mathcal{M}^{\mathcal{V}}\}$. □

We illustrate and summarize all kinds of mappings in Figure 3.3. In the figure, we see mappings from two sources as well as one join mapping. It shows that not all concepts or properties have to be supported by all sources. Global objects are combined by using an outer join operation that will be presented in the following section. If the system cannot find complementary local objects, a global object can be incomplete with respect to the global concept definition.

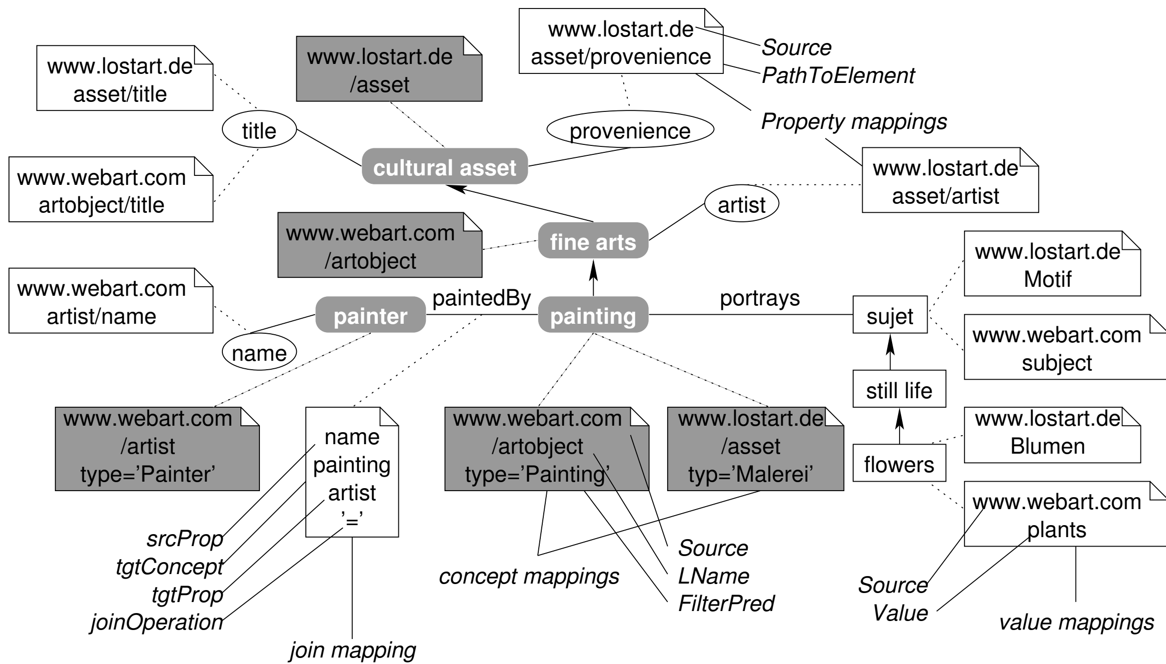


Figure 3.3.: Exemplary YACOB mapping

In summary, the mapping model uses features of GaV and LaV approaches. Concept mappings, property, and value mappings follow the LaV approach. That means, it is straightforward to add and remove new sources to the mediator system. Join mappings represent global views that implement intersource relationships. For this reason, join mappings represent the GaV approach in the YACOB integration approach. However, as join mappings are expressed using the global concept model, the problems of the GaV approach are mitigated. The system uses the mappings to translate concept queries into source queries as well as to transform local XML objects into global objects that conform to the global concept specification. For the second task, the YACOB system creates XSLT rules from the mappings and applies them to the local objects [SGS05].

Putting concept model, data model, and mapping model together, we can define the YACOB *integration schema*.

Definition 3.11 Integration Schema
The integration schema $I = (S, \mathcal{M}^{\mathcal{C}}, \mathcal{M}^{\mathcal{P}}, \mathcal{M}^{\mathcal{J}}, \mathcal{M}^{\mathcal{V}})$ *consists of the global schema* $S = (\mathbf{C}, \mathbf{P}, \boldsymbol{is_a}, \mathbf{V})$ *and other assigned mappings from the sources.* □

Based on the integration model and schema of the YACOB system, we will discus the query language as well as the query processing in the following sections.

3.2. Query Language

The YACOB system provides a concept-based query language named CQuery. CQuery supports operations on concept level, on instance or data level, and provides mechanisms allowing the transition between the both levels. We introduce the query language by means of representative examples. In the remaining section, we define the underlying query algebra and the translation of a CQuery statement into an algebra expression.

CQuery by example

CQuery is a derivative of XQuery [BCF+03], i.e., it follows the FLWOR[2] notation, but the semantics of CQuery differs from XQuery in many parts. The main new features of CQuery are on semantical level. A query expressed in CQuery consists of the following components:

1. selection of concepts based on conditions, path traversals, and set operations,
2. obtaining and filtering data as instances of selected concepts, and
3. combining and projecting the results.

Query Q_1 represents a typical query in CQuery (see Figure 3.4). Query Q_1 returns an XML document containing picture elements structured into title and artist's name representing paintings made by (van) Gogh. Concepts are selected in the `FOR` clause.

```
1 FOR $c IN concept[name="Painting"]
2 LET $e := extension($c)
3 WHERE $e/artist ~= "gogh"
4 RETURN
5   <picture>
6     <title>$e/title</title>
7     <artist>$e/artist</artist>
8   </picture>
```

Figure 3.4.: Listing CQuery query Q_1

Besides the concept selection used in query Q_1 other set operations on concepts are also supported, i.e., `UNION`, `INTERSECT`, and `EXCEPT`.

[2]FOR,LET,WHERE,ORDER BY,RETURN clauses in XQuery.

As concept schema defines relationships between concepts and categories, respectively, path expressions are supported by CQuery. For example, the expression

```
concept[name="Fine arts"]/!subClassOf
```

returns all concepts that are direct subclasses of concept `Fine arts`. The "`!`" denotes the inverse relationship of `subClassOf`. The suffix "+" specifies the computation of the transitive closure according to the given relationship. For example, the expression

```
concept[name="Fine arts"]/!subClassOf+
```

returns all concepts directly and indirectly derived from concept `Fine arts`. The shortcut "*" represents the path expression `!subClassOf+`. Using the shortcut, we can rewrite the expression above to `concept[name="Fine arts"]/*`. As the result of the `FOR` clause, an iteration over the selected concept set is bound to a variable.

For every concept bound to `$c`, the `LET` clause computes with the help of the function `extension()` the extension of the concept. The `FOR` clause represents an iteration over the selected concept set. The result set comprises global objects. It is bound to a new variable, in this case `$e`. Thereby, the variable `$e` iterates over the result set. This is a difference compared to the XQuery semantics.

The `WHERE` clause filters global objects using selection conditions. There, we can access properties using path expressions. Besides conventional comparison operators as =, <, <=, etc. a predicate may also contain a boolean text containment operator ∼=. For example, query Q_1 uses the condition `$e/artist∼="gogh"` to express that the string value in `$e/artist` has to contain the term "gogh". While operations of the `FOR` clause only use the concept level of the mediator system, the `extension()` function triggers the access to the sources to retrieve actual data objects as elements of the concept extensions. At last, the `RETURN` clause allows the projection and restructuring of the XML instances. CQuery assumes here, that any expression starting with a variable is a path expression to selected elements of the objects. The `RETURN` expression is applied to every object tuple created by the `LET` or `FOR` clause.

Besides simple selection, CQuery allows the join of instances. Q_2 represents a join query (Figure 3.5). Query Q_2 returns information about drawings and the location of their exhibition. The `WHERE` clause contains a join condition between two extensions of two different concepts. The result is a set of tuples comprising objects from `$e1` and `$e2`

CQuery allows in the `LET` clause further concept level operations. It is possible to retrieve properties of a concept and to bind them to a variable. In the `WHERE` clause, the variable is used to create a disjunctive query. Consider the exemplary query Q_3 in Figure 3.6. Query Q_3 retrieves all properties of the concept currently bound to variable `$c`. It binds the property set to the variable `$p`. The variable `$p` is a higher-order variable similar to query languages, e.g., SchemaSQL [LSS96]. The approach is implemented as follows. For example, assume the set `$p` contains the properties "title" and "artist". Then, the system rewrites the condition `$e/$p = "flowers"` into the disjunction `$e/title = "flowers" or $e/artist = "flowers"`. In this way, CQuery supports operations on properties, which are useful for queries with incomplete knowledge about the concept schema.

```
1  FOR $c1 IN concept[name="drawing"],
2      $c2 IN concept[name="collection"]
3  LET $e1 := extension($c1),
4      $e2 := extension($c2)
5  WHERE $e1/exhibition = $e2/museum
6  RETURN
7    <drawing>
8      <title>$e1/name</title>
9      <museum>$e2/name</museum>
10     <location>$e2/city</location>
11   <drawing>
```

Figure 3.5.: Listing CQuery query Q_2

```
1  FOR $c IN concept[name="painting"]
2  LET $e := extension($c),
3      $p := $c/properties
4  WHERE $e/$p = "flowers"
5  RETURN ...
```

Figure 3.6.: Listing CQuery query Q_3

As last concept of CQuery, we present the use of categories. The user can select categories using the `LET` clause. The result is bound to a variable and is used in selection conditions. Figure 3.7 shows an exemplary query statement. Query Q_4

```
1  FOR $c IN concept[name="graphics"]/*
2  LET $e := extension($c),
3      $k := $c/portrays[name="still life"]/*
4  WHERE $e/portrays = $k
```

Figure 3.7.: Listing CQuery query Q_4

returns all instances of concept `graphics` that show a “still life” as subject. In line 3, we assign the category hierarchy by using the path `$c/portrays`. Subsequently, we select the category “still life” and its sub-categories. The result is bound to the variable `$k`. In the `WHERE` statement, we use the set of categories. The system creates for every member `v` in `$k` a predicate `$e/portrays = v` and connects the predicates disjunctively.

Finally, the `RETURN` clause of CQuery is equivalent to the counterpart in XQuery and is used to project and restructure the results.

The query language CQuery has following features in comparison to other query languages in the context of RDF and Semantic Web:

- The semantics of concept level elements is retained, i.e., a concept is still a concept after applying a filter operation. The semantics preserving behavior of

CQuery simplifies concept level queries, which are necessary in data integration scenarios. Furthermore, it allows the interpretation of concepts as classes with extensions that are provided by remote sources.

- CQuery supports the concept level and the instance level as well as the transition from concept to instance level. On the one hand, the interpretation of instances as XML elements allows the usage of XML query features like element construction and transformation. On the other hand, CQuery provides high-level operations on the semantic level, e.g., transitive closure according to a property.

In summary, CQuery combines ideas of RDF query languages [HBEV04] and XQuery [BCF+03]. However, the language is closely related to classic multi-database query languages that support schema level queries as SchemaSQL [LSS96], FraQL [SCS03], or MSQL [KLK91].

CQuery algebra

The first step of query processing in YACOB comprises the query rewriting into an algebraic expression. We also define the semantics of CQuery with the help of the transformation into the algebra. We distinguish three classes of algebra operations: concept level operations, instance level operations, and level transition operation. First, we define the concept level operations.

Definition 3.12 Concept level operations
The ***concept level operations*** *of CQuery:*

Concept selection ($\Sigma : \mathbb{P}\mathcal{C} \rightarrow \mathbb{P}\mathcal{C}$): *The concept selection is defined for a set of concepts C and a condition F as*

$$\Sigma_F(C) = \{c | c \in C, F(c) \textit{is true}\}.$$

Path traversal ($\Phi : \mathbb{P}\mathcal{C} \rightarrow \mathbb{P}\mathcal{C}$): *Given a set of concepts C and a relationship p the path traversal returns the following set of concepts:*

$$\Phi_p(C) = \{c | \exists c' \in C : (p, c', c) \in \mathcal{P}\}.$$

The operation is also applicable for the inverse property $\overline{p}$:

$$\Phi_{\overline{p}}(C) = \{c | \exists c' \in C : (p, c, c') \in \mathcal{P}\}.$$

Transitive closure ($\Phi^+ : \mathbb{P}\mathcal{C} \rightarrow \mathbb{P}\mathcal{C}$)***:*** *Given a set C the operation Φ^+_p returns the transitive closure of a concept set regarding the relationship p:*

$$\Phi^+_p(C) = \{c | \exists c_s \in C : (p, c_s, c) \in \mathcal{P} \vee \exists c_i \in \Phi^+_p(\{c_s\}) : (p, c_i, c) \in \mathcal{P}\}.$$

In case of the inverse property $\overline{p}$ the result is

$$\Phi^+_{\overline{p}}(C) = \{c | \exists c_s \in C : (p, c, c_s) \in \mathcal{P} \vee \exists c_i \in \Phi^+_{\overline{p}}(\{c_s\}) : (p, c, c_i) \in \mathcal{P}\}.$$

Category selection ($\Sigma_F^{\mathcal{V}} : \mathbb{P}\mathcal{V} \to \mathbb{P}\mathcal{V}$)**:** *Let V be a set of categories, the condition F consists of a path consisting of a concept and a property and a constant selection predicate, then the category selection is defined as:*

$$\Sigma_F^{\mathcal{V}}(V) = \{v | v \in V \wedge F(v) \text{ is true}\}$$

Property selection ($\Sigma_F^{\mathcal{P}} : \mathbb{P}\mathcal{P} \to \mathbb{P}\mathcal{P}$)**:** *The property selection retrieves the literal and categorical properties for a concept set C and a condition F:*

$$\Sigma_F^{\mathcal{P}}(C) = \{p | p = (n, c, c') \in \mathcal{P} \wedge c \in C \wedge c' \notin \mathcal{C} \wedge F(p) \text{ is true}\}$$

Concept set operations: ($\cup, \cap, \setminus$)**:** *Sets of concepts can be combined by using following set operations:*

- *union* ($\cup$): $C_1 \cup C_2 = \{c | c \in C_1 \vee c \in C_2\}$
- *intersect* ($\cap$): $C_1 \cap C_2 = \{c | c \in C_1 \wedge c \in C_2\}$
- *except* ($\setminus$): $C_1 \setminus C_2 = \{c | c \in C_1 \wedge c \notin C_2\}$

□

The previous definition gives an overview about the operations on concept level. The concept level operations are used to represent the constructs of the `FOR` and `LET` clause. We will show the translation from CQuery to the algebra operation later on in the query processing section 3.3.

Transition operation

The transition from concept level to instance level is carried out by the `extension()` function of CQuery. The function is defined using the extension definition **ext**. However, most of the operators on instance level work on object tuples. An object tuple is a map $(a_1 : obj_1, \ldots, a_n : obj_n)$ where an alias $a_i \in Name$ indicates an object $obj_i \in \mathcal{O}$. The operator $\mathbf{ext}_a : \mathcal{C} \to \mathbb{P}\mathcal{O}_1^T$ ($\mathcal{O}_1^T$ denotes the set of all object tuples of the arity 1) with result tuples of the form $(a : o)$. The parameter a of $\mathbf{ext}_a$ denotes the alias. The notation $var(t)$ returns all aliases in t. Given a tuple $t = (a_1 : obj_1, \ldots, a_n : obj_n)$ the notation $t(A)$ with $A \subseteq var(t)$ describes the projection of t using only the entries in A, i.e., $t(A) = (a_{i1} : obj_{i1}, \ldots, a_{in} : obj_{in})$ with $a_{ij} \in A$ and $n = |A|$.

On an object o, we have the following operations:

- $properties(o)$ returns the set of properties in o and
- $o(p)$ returns the object value of property p.

Furthermore, we can merge two objects. We assume, we can merge two objects with overlapping property sets if the common properties have the same values. Let o and o' be two objects with overlapping properties, i.e., $properties(o) \cap properties(o') \neq \emptyset$. We require that for all $p \in properties(o) \cap properties(o')$ that the objects have the same value,i.e., $o(p) = o'(p)$. In this case, the operation $o'' = merge(o, o')$ creates an object o'' with the values: for $p \in properties(o) : o''(p) = o(p)$ and for $p \in properties(o') : o''(p) = o'(p)$. The operation *merge* returns `null` if the conditions have not been satisfied.

Instance level operations

After the transition to the instance level, the object tuples are filtered, combined, and restructured using further operations. We restrict the discussion here to the operators selection (σ), projection (π), join ($\bowtie$,) and extensional union $\uplus$. The `RETURN` clause can be implemented using element construction and tree projection operators. However, the focus of this work lies on the integration and creation of global objects and object tuples without further processing. Following, we define the standard operators in detail.

Definition 3.13 Instance level operations
*The **instance level operations** are:*

Selection ($\sigma_F : \mathbb{P}\mathcal{O}_n^T \rightarrow \mathbb{P}\mathcal{O}_n^T$): *The instance selection on a set O of object tuples and a given condition F is defined as*

$$\sigma_F(O) = \{t | t \in O, F(t) \text{ is true}\}.$$

Projection ($\pi_P : \mathbb{P}\mathcal{O}_m^T \rightarrow \mathbb{P}\mathcal{O}_n^T$): *Let O a set of object tuples and $P = \{a'_1, \ldots, a'_n\}$ a set of aliases, then the projection $\pi_{\{a'_1,\ldots,a'_n\}}(O)$ is defined as*

$$\pi_P(O) = \{t(P) | t \in O\}$$

Join ($\bowtie_F : \mathbb{P}\mathcal{O}_n^T \times \mathbb{P}\mathcal{O}_m^T \rightarrow \mathbb{P}\mathcal{O}_{n+m}^T$): *The instance join combines two object tuple sets by concatenating all pairs and returning object tuples that satisfy the condition F:*

$$O_1 \bowtie_F O_2 = \{t | \exists t_1 \in O_1 \exists t_2 \in O_2, t = concat(t_1, t_2) \wedge F(t) \text{ is true } \}.$$

Extensional Union ($\uplus : \mathbb{P}\mathcal{O}_1^T \times \mathbb{P}\mathcal{O}_1^T \rightarrow \mathbb{P}\mathcal{O}_1^T$): *The extensional union merges two sets of objects to integrated objects:*

$$O_1 \uplus O_2 = \{(a : o) | (a : o_1) \in O_1 \wedge (a : o_2) \in O_2 \wedge (o = merge(o_1, o_2)) \neq \texttt{null}\}$$

□

The extensional union merges objects of two sets assigned to the same variable. The operation belongs to the class of outer joins. For example, objects coming from different sources are combined in this way. It does not handle data level integration conflicts.

After defining the algebraic operators, we summarize the rules for transformation of CQuery expressions to the algebraic expressions. Let $S = (\mathbf{C}, \mathbf{P}, is_a, \mathbf{V})$ be the global schema. The transformation rules are:

1. The clause `FOR $c IN concept[Cond]` is translated into

$$\Sigma_{Cond}(\mathbf{C}).$$

2. The statement `FOR $c IN concept[Cond]/*` is transformed into the expression

$$\Phi^{+}_{\overline{\mathbf{is_a}}}(\Sigma_{Cond}(\mathbf{C})),$$

where $\overline{\mathbf{is_a}}$ represents the inverse **is_a** relationship.

3. Path traversal expressions like `concept[Cond]/prop`$_1$`/.../prop`$_n$ are translated into

$$\Phi_{prop_n}(\ldots(\Phi_{prop_1}(\Sigma_{Cond}(\mathbf{C})))).$$

4. A clause `LET $p := $c/properties[Cond]` is transformed into the expression

$$\Sigma^{\mathcal{P}}_{Cond}(C)$$

where C represents the set of concepts bound to variable c.

5. A set of categories is selected using the clause `LET $k := $c/p[Cond]`. The clause is translated into the expression:

$$\Sigma^{\mathcal{V}}_{cond'}(\mathbf{V})$$

with $cond' = \$c/p \wedge Cond$.

6. The previous operations operate on concept level. The clause `LET $e := extension($c) WHERE Cond` is translated as follows. Let $CExpr$ be the concept expression bound to variable `$c`, then the statement is transformed into

$$\biguplus_{c \in CExpr} \sigma_{Cond}(\mathbf{ext}_e(c)).$$

If more than one extension variables have been defined, the Cartesian product is computed between these sets.

7. A path expression in `WHERE` or `RETURN` that contains a concept property is rewritten into a join operation using join mapping information. For instance: the selection

$$\sigma_{rel/p=A}(\mathbf{ext}_e(c))$$

uses a path expression containing the concept property rel. Assume, there is a join mapping $JM = (p_{src}, c_{tgt}, p_{tgt}, =)$ for rel, then the selection is translated into expression

$$\pi_{(e)}\left(\sigma_{e'/p=A}\left(\mathbf{ext}_e(c) \bowtie_{p_{src}=p_{tgt}} \mathbf{ext}_{e'}(c_{tgt})\right)\right).$$

8. The `RETURN` clause is translated into a return operator $RETURN$ consisting of element constructions.

We now describe the transformation rules by means of an example. Assume query Q_5 (Figure 3.8): The query Q_5 is translated into the expression

$$\pi_{(e)}\left(\biguplus_{c_1\in CExpr_1} \sigma_{Cond}(\mathbf{ext}_e(c_1)) \bowtie_{artist=name} \biguplus_{c_2\in CExpr_2} \sigma_{born<1800}(\mathbf{ext}_{e2}(c_2))\right),$$

with

$$\begin{aligned} CExpr_1 &= \Sigma_{name='painting'}(\mathbf{C}) \\ CExpr_2 &= \Sigma_{name='painter'}(\mathbf{C}) \\ Cond &= \bigvee_{k\in\Sigma^{\mathcal{V}}_{name='\text{still life}'\wedge c_1/portrays}(\mathbf{V})} portrays = k \end{aligned}$$

by using the following steps. Rule 1 is applied to translate the `FOR` clause to concept expression $CExpr_1$. Property `paintedBy` is an inter-concept relationship with attached join mapping $JM = (artist, painter, name, =)$. Therefore, the path expression `paintedBy/born` is rewritten into a join between instances of the concept `painting` and instances of the concept `painter`. As categories are referenced in the `WHERE` clause, the expression `$k := $c/portrays[name='still life']` is translated into the expression $\Sigma^{\mathcal{V}}_{name='\text{still life}'\wedge c_1/portrays}(\mathbf{V})$. Subsequently, the condition `$e/portrays = $k` is rewritten into the disjunctive condition $Cond$. The `RETURN` clause is translated into the element construction.

```
1  FOR $c IN concept[name="painting"]
2  LET $e := extension($c)
3      $k := $c/portrays[name="still life"]
4  WHERE $e/paintedBy/born < 1800 AND $e/portrays = $k
5  RETURN
6    <painting>
7      <title>$e/title</title>
8      <artist>$e/artist</artist>
9      <motif>$e/portrays</motif>
10   </painting>
```

Figure 3.8.: Listing CQuery query Q_5

3.3. Query Processing

The translation of a CQuery statement into an algebraic expression is the first step of the query processing. The result are expressions of the form $\biguplus_{c\in CExpr}(IExpr(c))$. If the original query has been a join query, several blocks of such expressions have to be executed. The next step is the rewriting of these expressions into source queries. Source queries are simple XPath queries that are supported by local sources, like

Web sources. Algorithm 3.1 shows the corresponding algorithm [SGS05], which uses a semantic cache [KSGH03, Kar03].

Algorithm 3.1 Steps of query processing [SGS05]

```
    Input:
      query expression of the form of ⊎_{c∈CExpr} IExpr(c)
      result R := {}

 1: function QueryProc(⊎_{c∈CExpr} IExpr(c),R)
 2:    compute concept set C := CExpr
 3:    for all c ∈ C do
 4:       /* translate query into XPath */
 5:       q := toXpath(IExpr(c))
 6:       /* look for query q in the cache → result is denoted by R_c, q̄ is returned as
   complementary query to q */
 7:       R_c := CacheLookup(q, q̄)
 8:       if R_c ≠ ∅ then
 9:          /* found cache entry */
10:          R := R ⊎ R_c
11:          q := q̄
12:       end if
13:       if q ≠ empty then
14:          /* derive source query */
15:          for all CM_s associated with c do
16:             s := CM_s(c).Source
17:             /* consider only non-redundant concepts */
18:             if c ∈ cmin(C, s) then
19:                /* construct a query for each supporting source c: determine all
   properties p_i(i = 1...n) and categories k_j(j = 1...m) referenced in IExpr */
20:                q_s := TranslateForSource(q, CM_s(c), PM_s(p_i), VM_s k_j)
21:                R_s := ProcessSourceQuery(q_s, s)
22:                R := R ⊎ R_s
23:             end if
24:          end for
25:       end if
26:    end for
27: end function
```

Initially, concepts are selected by applying the expression *CExpr* on the concept schema. Secondly, the system evaluates for each concept c the instance expression $IExpr(c)$. The instance expression is translated into an XPath query. Thirdly, the system does a cache-lookup in order to utilize previous results. The semantic cache returns the (partial) result and if necessary a complementary query $\overline{q}$, which retrieves the complementary data to a partial result. Details of the function CacheLookup are given in [KSGH03, Kar03].

If the cache does not deliver an answer or only a partial result, then the system will translate the global query for each supporting source s and send local queries to the particular sources. Redundant concepts have to be eliminated, due to possible overlapping between different concepts in one source. This step is performed using function $\text{Cmin}(C, s)$ that returns the minimal, non-redundant subset of concepts for given source s and concept set C. The computation is based on the heuristics *Elimination* and *Merging*. Concepts are eliminated from C if they are a sub-concept of another concept in C and they are mapped to the same local element name in a source. Two concepts will be merged if they are mapped to the same local element name. During the merging step, the filter predicates of the respective concept mappings have to be connected disjunctively.

In the next step, the remaining subqueries are translated into source queries using the mapping information of concepts, properties, and categories. For example, assume the query $\sigma_{P=v}(\mathbf{ext}(c))$ and the given concept mapping $(cm \to c) \in CM(c)$ and the property mapping $(pm \to p) \in PM(p)$. The source XPath expression is built in the following way:

$$//cm.LName[./pm.PathToElement = v \text{ } and \text{ } CM.FilterPred].$$

If v represents a category, the category mapping $(vm \to v) \in VM(v)$ is used to translate the category into local data values. Thereby, it must hold that cm, pm, and vm are mappings of the same source s.

In summary, Algorithm 3.1 describes single concept query processing in the Yacob system. It optimizes queries based on concept hierarchies in one source and a single concept semantic cache. The result consists of simple XPath queries. These queries are sent to the sources through wrappers. The wrappers allow that the XPath queries can be executed by Web sources, as well as XML and relational database management systems. Sattler et al. investigate the performance of the approach [SGS05]. In this way, the Yacob system provides the selection and integration of objects from heterogeneous sources. We will extend the approach with join processing to answer keyword queries in Chapter 7.

3.4. Summary

We described the Yacob mediator system in this chapter. The system is based on a concept-based integration schema. Yacob uses a LaV approach for its schema mappings. Thereby, the schema mappings follow the RDF approach by mapping properties and concepts separately. The concept model expresses relationships between concepts with the help of concept properties. The concept properties are mapped to global join definition. While this is a GaV approach, it is mitigated by using only global defined concepts and properties. The Yacob integration model maps directly the local XML elements to the global concepts. Thus, it is similar to Amann et al. [ABFS02b]. In contrast, the Yacob integration model uses only concept and property mappings and does not use local relationships. However, Yacob covers also categorical values. The presented language CQuery is inspired by XQuery as well

as classic multi-database languages. The query evaluation optimizes single concept queries and allows string containment selection conditions. The mediator YACOB is suitable to access Web sources, simple XML sources, and relational databases in a unified manner. We use YACOB as foundation of the keyword search system presented in the following chapters. It answers single concept queries and is used by global keyword and join processing algorithms.

4. Keyword Search in Structured Databases

Global integration schemas, concept-schemas, or ontologies allow a flexible integration, but, also suffer from problems in usability because of complex and evolving data structure, unknown content, and complex global or local query languages. Similar problems arise in centralized, (semi-)structured databases, too [JCE+07]. In the current chapter, we review the research in the area of keyword search in structured databases. This review complements other surveys [CWLL09, YQC10, WZP+07] by adding a focus to distributed and heterogeneous systems. After an introduction and general definition in Section 4.1, a classification of keyword search approaches in (semi-)structured database is presented in Section 4.2. In Section 4.3, we describe several exemplary keyword search systems. The problem of keyword search in distributed, heterogeneous, structured databases is discussed in Section 4.4. The chapter concludes with a summary in Section 4.5.

4.1. General Considerations

Nowadays, a huge amount of information is stored in database systems. Normally, database systems are structured according to a database model. Typical database models are variants of the relational data model or XML. Database management systems provide an excellent support for efficient storage, management, and querying of this data. Structured data allow exact and detailed search over DBMS support complex and powerful query languages as SQL [MS02] and XQuery [BCF+03] in order to access the data. Because of these facts and the following points, databases tend to be hard to use (following [JCE+07]):

- Databases are complexly structured. Real world objects are split into different elements due to normalization in order to avoid redundancy, update anomalies, etc. The relationships between single elements may be also manifold: e.g., containment relationships, foreign key relationships, or id/idref relationships. Furthermore, the schema may also evolve, for instance in semi-structured or object-oriented databases [KS01]. In this case, the user cannot learn the structure at all.
- While complex structures may be a burden for the user, the structure also contains data semantics. Thus, if the user has any (partial) knowledge about the database structure, the system has to provide query support for it. The problem is that partial structural information leads to empty or false results in traditional, structured query languages.

- Users have an expectation of structured results, e.g., tuple trees, XML elements or trees, connection graphs, etc. While traditional search engines need to give only link lists as an answer, databases have to support detailed, structured answers.
- During querying databases, the user has strong assumptions about the content of the database and wants to have exact and complete results. Thus, the requirements are much stronger compared to web search engines, where users have only a diffuse knowledge about the content.
- Further problems relate to designing and building a database, i.e., a user does not know exactly in which direction the content of a database and the corresponding schema will evolve in the future. Another point is the explanation of a result. The user wants to know why a result is as it is. Furthermore, the user is interested in which sources are used to construct the result. That point relates to the data provenance problem.

The first three points concern to the presentation layer of the database system, because the actual logical data models are not suitable in every case for query construction and result presentation. Several earlier works dealt with that problem. On the one hand, visual query languages were developed, for instance, the seminal work of Query by example by Zloof [Zlo75]. On the other hand, also new textual query languages were proposed in order to hide complex structural relationships. A classic concept is the universal relation [Ull90]. The universal relation inspired several keyword search approaches presented in the remainder of the chapter.

If we take these points into account, it is necessary to combine concepts of information retrieval and database retrieval [CRW05].

4.1.1. Example

In order to describe requirements and exemplary use cases, we present an exemplary database. Figure 4.1 illustrates a data graph of a (semi-)structured database storing information about cultural assets. Four basic elements exist: `cultural_assets` containing different kinds of cultural assets (e.g., paintings, drawings, sculptures), `artists`, `epochs`, and `institutions`. Besides containment relationships, the database provides further relationships between cultural assets and artists, representing a "created by" relationship, and between cultural assets and epochs that classifies assets into an epoch. Based on that data graph, users may issue different kinds of complex queries. Several typical queries are described in the following example.

Example 4.1 *Based on the data graph illustrated in Figure 4.1 following exemplary queries may be issued by users:*

Query 1: Find paintings made by ***Vincent van Gogh****.*

Query 2: Find artists and their cultural assets, which belong to the epoch ***renaissance****.*

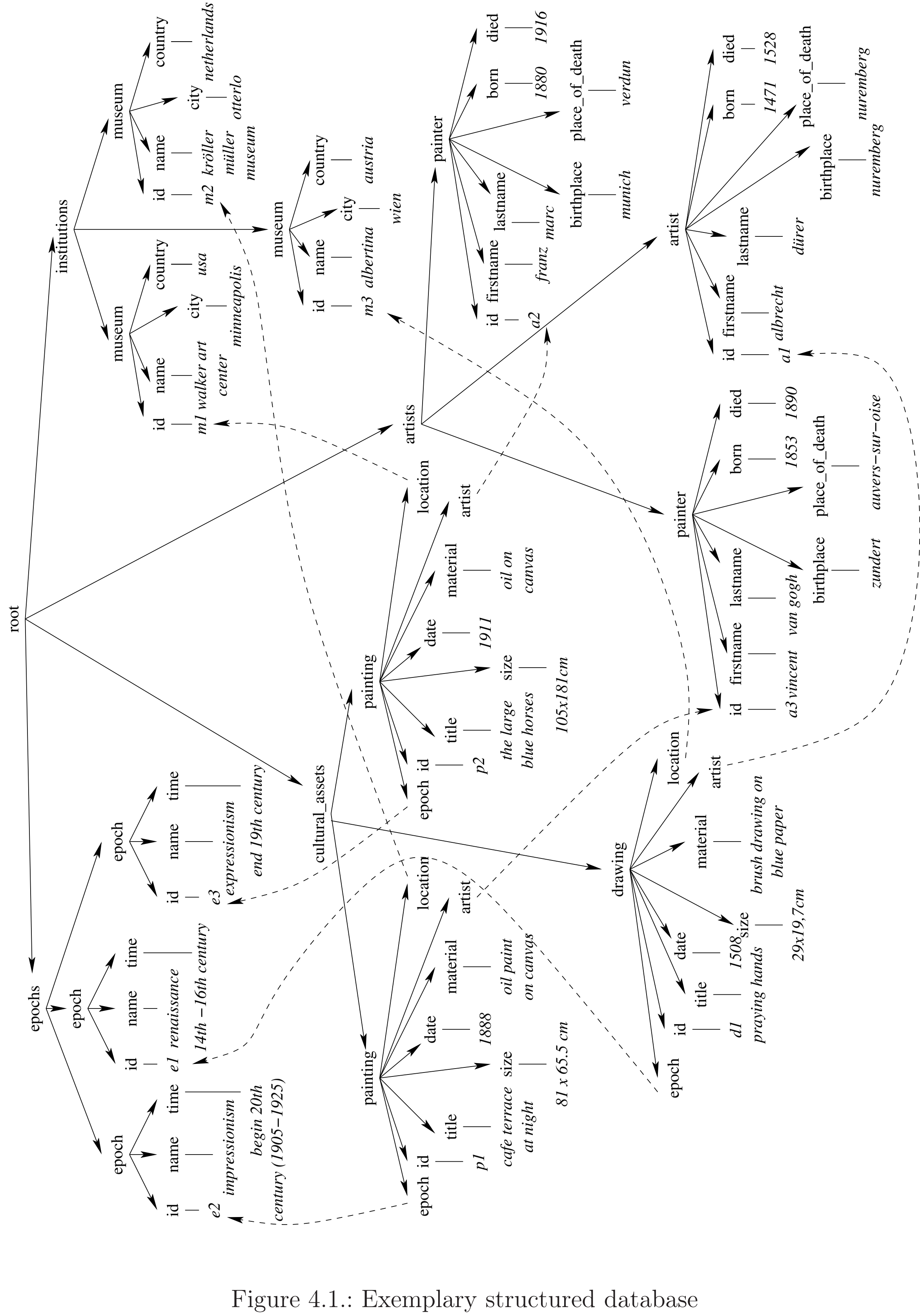


Figure 4.1.: Exemplary structured database

*Query 3: Find artists, which are shown in **Minneapolis USA**.*

*Query 4: Find museums exhibiting **Gogh** paintings.*

*Query 5: Find everything for **Gogh Netherlands**.*

*Query 6: Find the title and year of paintings of painter with name **Dürer**.*

*Query 7: Find artists that exhibited in the same institutions as paintings by **Vincent van Gogh**.*

The queries represent different use cases. Besides simple keyword queries like Query 5, there are different levels of structural information given by the user. For instance, Query 1 searches for paintings by Vincent van Gogh, while Query 6 specifies exactly title and year of paintings as well as that the name of the painter is searched.

4.1.2. Definitions

Now, we give several general definitions in order to describe following approaches in a common way, which are inspired by Yu and Jadagish [YJ06].

Definition 4.1 Terms and Labels
The set of all terms occurring in a document is denoted as $\mathcal{T}$. The set of all labels, names for nodes, is denoted as Name. □

The set $\mathcal{T}$ represents all terms that are found in textual values of nodes of a document. For example, in Figure 4.1 all values presented in *italics* are textual values. Labels are names of nodes or edges. The structure of a database is described by a schema. In this work, the schema graph is described as a directed graph SG.

Definition 4.2 Schema graph
A schema graph is a directed graph $SG = (N_S, E_S)$. $N_S \subset Name$ is the set of nodes, which represent the labels in a database. The set of directed edges $E_S \subseteq N_S \times N_S$ describe the relationships between two schema nodes. Furthermore, a function $\lambda_{E_S} : E_S \to Name_E$ assigns labels to an edge with $Name_E \subset Name$. □

A schema graph can be constructed from a data graph, e.g., a data guide [GW97], or be defined manually as relational and XML schema. Figure 4.2 shows one exemplary schema graph for the data graph of Figure 4.1. Databases or semi-structured documents are described as data graph D.

Definition 4.3 Data graph or document
A data graph or document is a labeled, directed graph $D = (N_D, E_D)$ with N the set of nodes and $E_D \subseteq N_D \times N_D$ the set of edges in the graph. A label function $\lambda_N : N_D \to Name_N$ assigns a label $l \in Name_N \subset Name$ to a node $n \in N_D$, and function $\lambda_E : E_D \to Name_E$ labels an edge with an edge label. □

Figure 4.1 illustrates a data graph example for an art database.

Besides labels, nodes and edges may also have types. The following definition specifies corresponding functions and sets.

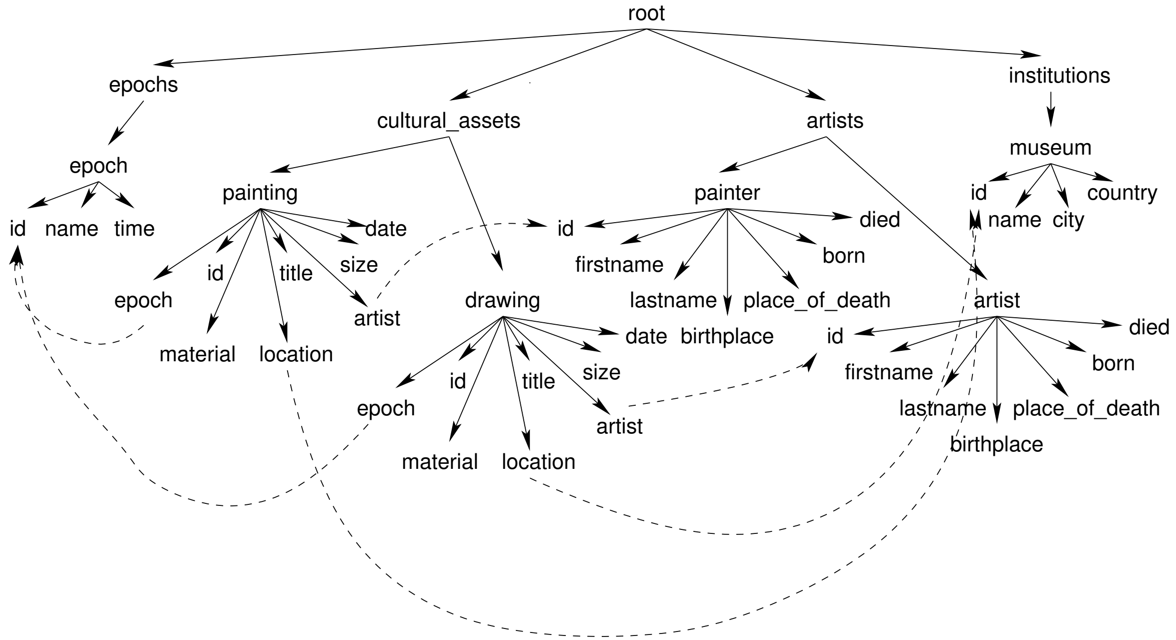


Figure 4.2.: Schema graph example

Definition 4.4 Structural node and edge types.
The set of all node types is denoted as NType, the set of edge types is denoted as EType. To assign node and edge types the following two functions are defined:

- $\textbf{\textit{type}}_N : N_D \rightarrow NType$ *assigns a type to a node in a data graph,*
- $\textbf{\textit{type}}_E : E \cup E_S \rightarrow EType$ *assigns an edge type to an edge in the schema or data graph.*

□

The nodes in a schema graph are always labels, which describe the semantic type of the node. Typical structural types of nodes in data graphs are structural nodes, e.g., internal XML elements or tuples, relations, and attributes in relational databases. Content nodes are leaves in the data graph and contain the actual data values.

Edge types are distinguished into two main types: containment edges and link edges. Containment edges describe containment or nesting hierarchies. For example, they represent the nesting of XML elements or the hierarchy relation - tuple - attribute - attribute value. Link edges stand for a different kind of reference relationships, e.g., id/idref or key/keyref relationships in XML documents, XLinks/XPointer as inter-document dependencies, or foreign key relationships in relational databases. Let $CNT, LNK \in EType$ be two edge types, than the set of containment edges is defined as $E_{cnt} = \{e|e \in E_D \wedge \textbf{type}(e) = CNT\}$. The set of link edges is specified as $E_{lnk} = \{e|e \in E_D \wedge \textbf{type}(e) = LNK\}$. Furthermore, it holds: $E_{cnt} \cap E_{lnk} = \emptyset$.

A data graph DG conforms to a schema graph SG if the nodes are labeled according to the schema graph and the edges conform to corresponding edges in the schema graph.

Definition 4.5 Schema graph conformance.
A data graph $D = (N_D, E_D)$ conforms to a schema graph $SG = (N_S, E_S)$, denoted as $D \models SG$, if

1. $\forall n \in N_D \exists n_s \in N_s : \lambda_N(n) = n_s$
2. $\forall (n, n') \in E_D \exists (n_s, n'_s) \in E_S : \lambda_N(n) = n_s \wedge \lambda_N(n') = n'_s \wedge \lambda_E((n, n')) = \lambda_{E_S}((n_s, n'_s))$,
3. $\forall (n, n') \in E_D \exists (n_s, n'_s) \in E_S : \lambda_N(n) = n_s \wedge \lambda_N(n') = n'_s \wedge \boldsymbol{type}_E((n, n')) = \boldsymbol{type}_E((n_s, n'_s))$.

□

In the definition, rule 1 ensures that the label of a node is in the schema graph. Rule 2 states that the edges of a data graph are labeled according the edges in the schema graph. Rule 3 ensures that the structural type of an edge conforms to the specified type in the schema graph.

4.2. Classification

We now present classification categories for keyword search approaches in structured databases. In detail, the classification dimensions comprise

- the query language,
- the result type,
- the scoring and ranking approach, and
- the query processing approach.

4.2.1. Query Languages

Different user groups have different levels of knowledge about database content and structure as well as query languages. While experts prefer complex query languages, less experienced users may want to use plain keyword queries. Besides different demands, technical reasons lead to different query types, too. Following [AYL06, JCE+07], query languages can be classified into the groups:

Keyword-only query: A keyword-only query consists of a set of keywords using *AND* or *OR* semantics, respectively. A query does not contain any structural constraints. Thus, the query is expressed as $Q = \{kw_1, kw_2, \ldots, kw_n\}$ with $kw_i \in (Name \cup \mathcal{T}), 1 \leq i \leq n$.

Label and keyword query: Labeled keyword queries are a set of pairs consisting of a label and a keyword. The result nodes have to be labeled with respect to the query label and have to contain the keywords. Given the set of labels *Name* and the set of terms $\mathcal{T}$, a label-keyword query is a set $Q =$

$\{(l_1, kw_1), (l_2, kw_2), \ldots, (l_n, kw_n)\}$ with $(l_i, kw_i) \in Name \times \mathcal{T}$ for $1 \leq i \leq n$. Both query types, keyword-only and label-keyword, are structure-free, i.e., a user cannot specify the structural constraints between the nodes.

Path and keyword query: Elements are selected by structural information using path queries. Thus, a natural extension is the combination of path and keyword queries. A path query consists of location steps [xpa07]. Each location step consists of an axis specification, a node test and an optional predicate. Let p be such a location path and $Q_{kw} = \{kw_1, kw_2, \ldots, kw_n\}$ with $kw_i \in \mathcal{T}$ be a keyword query, then the pair (p, Q_{kw}) is a path-keyword query. Here, the keyword expression is a predicate in a location step. Different path-keyword queries can be combined to complex queries. Furthermore, node test operations can be made flexible by using ontology approaches [TW02a, LYJ04]. An example query in XPath plus fulltext extension is `//artists/painter[. ft:contains "vincent"]`.

Full query language and keyword query: The complete integration of structured and unstructured queries is the combination of a full-fledged structured query language (e.g., SQL or XQuery) and keyword search. Thereby, two problems have to be considered:

1. How to relax structural query constraints?
2. How to integrate seamlessly keyword queries into the query language?

Relaxed-structured queries allow the user to give structural information without penalizing the user for it, e.g., [KS01, AYLP04, LYJ04]. Schema-free XQuery [LYJ04] is one language proposal including relaxed structural queries.

The second problem is the integration of a keyword search and structured query language. For the standard language XQuery, a full text search extension exists [AYBDS06], which allows the seamless integration. Here, results of keyword searches can be further manipulated by XQuery statements, and vice versa: XQuery results can be used in the full text query parts.

Natural language interface: The last kind of textual interfaces considered here is the natural language interface. In this case, the query consists of natural language, e.g., sentences in written English. Subsequently, the system analyzes the sentence and tries to map the language tokens to a keyword query with structural information. An example of this paradigm is the Nalix system [LYJ07, LCY+07].

4.2.2. Query Result Types

In structured databases, real-word data is spread over several data items. That means, data is stored in different tuples of a relational system or in different elements in an XML database. The problem of keyword search in structured databases is the definition of meaningful results with respect to a query.

Example 4.2 *Consider the document in Figure 4.1. Let* `Vincent` *and* `Munich` *be a keyword query. The element* `artists` *contains both keywords, but it is not a meaningful result, because it contains all artists only. Secondly, assume the keyword query*

$\{franz, marc, minneapolis\}$. A meaningful answer would be a graph containing the painter object $a2$, the painting $p2$, and the institution $m1$, because Franz Marc is the painter of the painting "The large blue horses" that is exhibited in Minneapolis.

Because of these issues, it is necessary to define what meaningful results to a query are. First, we describe the result of relational keyword search systems, second, we review several XML tree-based results, and third, we give result types for general graphs.

Relational model. A relational database schema can be seen as a graph of relation names and edges between them. The edges represent the foreign key relationships and are link edges in the proposed terminology. The basic data items are tuples. That means, a tuple t of a relation $r(R)$ contains a keyword kw, if any attribute value $t(A)$ contains kw. The result of a keyword query $Q = \{kw_1, kw_2, \ldots, kw_n\}$ is a **minimal total joining network of tuples** [HP02]. Two tuples $t_1 \in r(R_2)$ and $t_2 \in r(R_2)$ are connected, if they can be joined based on an edge between R_1 and R_2 and the corresponding foreign key relationship. In a joining network of tuples, all adjacent tuples are connected. The network is *total* if all keywords of Q are contained in the network, i.e., every keyword is contained in at least one tuple. The network is *minimal*, if no tuple (or edge) can be removed without breaking the other two properties. That induces, that leaves[1] have to contain keywords.

XML model. In XML documents, we have a tree structure, i.e., data items are contained by other data items. Our notation uses edges of type CNT to describe the containment relation. In the abstract description, data graph items are either labeled, internal element nodes or data content nodes. Different definitions of meaningful results with respect to Q in data trees exist. The basic definition is the *lowest common ancestor* $lca(N)$ of a set of nodes. The $lca(N)$ is a common ancestor of all nodes in N, and there does not exist a descendant n' of $lca(N)$ that is a common ancestor, too. The set $lca(N_1, N_2, \ldots, N_n)$ is the set of lowest common ancestors of all combinations of nodes from the node sets $N_1, N_2, \ldots, N_n$ [YQC10]. Based on these basic definitions further restrictions have been defined like the set of smallest lca nodes $slca(N_1, N_2, \ldots, N_n)$ [XP05] or the set of exclusive lca nodes $elca(N_1, N_2, \ldots, N_n)$ [GSBS03]. A summary of these classes is given in [YQC10, XP08]. Assume the keyword query $Q = \{kw_1, kw_2, \ldots, kw_n\}$ and let $N_1, N_2, \ldots, N_n$ be node sets with all nodes $n_i \in N_i$ contain the keyword kw_i. Yu et al. define the result of Q for a given XML tree T the result with respect to Q as follows: The result is a set of subtrees of T that contain all keywords. A subtree (t, M) is represented by its root node t and set of nodes M that contain the keywords directly. The nodes in M are denoted as matching nodes. The node t is lca of the nodes M [YQC10].

Different definitions of meaningful result subtrees exist. Following approaches are distinguished:

Smallest lowest-common ancestor (slca): Given is a keyword query Q, a result of the query is a node $n \in N$ with $\forall kw \in Q : contains(n, kw)$[2], and there is

[1] A leaf tuple has exactly one adjacent, connected tuple.

[2] Node n contains the keywords directly or indirectly.

no node n' with $\forall kw \in Q : contains(n', kw)$ and n' is a descendant of n. If $N_1, N_2, \ldots, N_n$ are node sets whose nodes directly contain the corresponding keywords $kw_1, kw_2, \ldots, kw_n$, the node n is an element of the set of smallest LCA $slca(N_1, N_2, \ldots, N_n)$. That means, the result tree or its root node cannot contain a subtree or a descendant that also contains all keywords. There are systems that return only the root node of the result trees [FKM00], others return the complete tree [XP05].

Exclusive lowest common ancestor (elca): The previous definition does not allow nodes that contain another node, which already satisfies all keywords. The exclusive lowest common ancestor semantics allows a node to be an answer that contains all keywords after removing sub-elements that contain all keywords. For example, if we assume the database in Figure 4.1 and keyword query $\{blue, canvas\}$. A valid result would be the painting p_2. But, the node "cultural_assets" is also a result, because after removing p_2 from this sub-tree, the keywords can be found in the painting p_1 and in the drawing $d1$. The smallest lca semantic would not allow this answer. XRANK [GSBS03] and [XP08] use the elca semantics.

Interconnected Node Sets: Cohen et al. [CKKS05, CMKS03] introduced the idea of interconnected nodes sets. The idea is, the matching nodes M of a result tree (t, M) must be *meaningfully related*. Below we give an exemplary meaningful relationship. There are two kinds of results:

1. in the node set M, all nodes are pairwise interconnected, or
2. in the node set M, one node exists that is connected to every other node in the set M.

An exemplary meaningful relationship between two nodes n_1, n_2 in M is defined as follows. Let $p(n_1, n_2)$ be the combined path from n_1 to $lca(n_1, n_2)$ and from n_2 to $lca(n_1, n_2)$. The nodes n_1 and n_2 are interconnected, if first, there are no two distinct nodes with the same label in $p(n_1, n_2)$, or second, there are exact two distinct nodes, the nodes n_1 and n_2, with the same label. For example, search for $\{minneapolis, amsterdam\}$ does not result into the node *museums* because of the paths are two distinct nodes with the label *museum*. However, assume a root node *book* and there are two author nodes as children that directly contain the keywords. In this case, the *book* sub-tree is a valid result.

Meaningful Lowest Common Ancestor Structure: Li et al. introduced the meaningful lowest ancestor sets (MLCAS) as query results in XML data trees [LYJ04, LYJ08]. For this, Li et al. extended the XQuery language to Schema-free XQuery. In the discussion, it means the matching nodes sets $N_1, N_2, \ldots, N_l$ have also a given semantic type, i.e., the nodes $n_i \in N_i$ have all the type A_i. For example, the nodes have a common label. Li et al. [LYJ08] define the set $mlca(N_1, N_2)$ of meaningful lowest common ancestors as:

1. for all nodes $c \in mlca(N_1, N_2)$ exists at least one pair of nodes (n_1, n_2) with $n_1 \in N_1$ and $n_2 \in N_2$ and $c = lca(n_1, n_2)$, and

2. for all pairs (n_1, n_2), the corresponding node $lca(n_1, n_2)$ is in $mlca(N_1, N_2)$ or there exists a node $n' \in mlca(N_1, N_2)$ and n' is descendant of $lca(n_1, n_2)$.

The idea of Li et al. is the following. Two nodes of distinct types are meaningful connected if they are in ancestor-descendant relationship, or if they have a common ancestor. The exception to this rule is described in the following example. Consider the nodes n_1 with label "firstname" (of painter $a2$), n_2 with label "lastname" (of artist $a1$) in Figure 4.1, and n_3 with label "lastname" (of artist $a2$). The lca of n_1 and n_2 is the node "artists", while the lca node of n_1 and n_3 is the node "artist" $a1$. Therefore, n_1 and n_3 are considered more closely related as n_1 and n_2 and their lca is in the set $mlca$ and $lca(n_1, n_2)$ is not. The $mcla$ semantics is based on the smallest lca semantics [YQC10] but takes into account the semantic types of the nodes. Li et al. also extended the definition to l sets [LYJ04, LYJ08].

Minimum Connecting Trees: Given are a tree document $DG = (N, E)$ and a set of nodes $\{n_1, n_2, \ldots, n_k\}$. The minimum connecting tree (MCT) is a subtree of DG that connects all nodes $\{n_1, n_2, \ldots, n_k\}$. The root of the tree is the lowest common ancestor of $\{n_1, n_2, \ldots, n_k\}$ [HKPS06]. Thus, this approach is an extension to the search of LCAs in such way, that the structure of the connections can be analyzed and ranked. This approach is similar to minimal joining networks of tuples.

Schema-based Matching: Yu and Jagadish [YJ06, YJ07] also investigated meaningful related elements in XML documents. The authors extended the tree-based definitions to XML tree with value relationships, i.e., link edges. In a schema, an element e_A is denoted as general parent of another e_D if e_A contains the element e_D or e_A is referred by e_D with a value link. The corresponding definitions are general ancestor and general descendant. For example, the element "location" of a drawing is a general descendant of the element "museum" because it refers to the "id" element of museum. Note, the elements "drawing" and "museum" have a common descendant "location" (see schema graph in Figure 4.2) now. Yu and Jagadish consider keywords of the form $label : keyword$. Therefore, the definition is based on meaningful schema pattern and meaningful data fragments.

Given a labeled keyword query $Q = \{l_1 : kw_1, l_2 : kw_2 : \ldots : l_n, kw_n\}$ with $L(Q)$ the label set. A meaningful schema pattern $P = (N_p, E_p)$ is a connected subgraph of the schema graph SG with respect to $L(Q)$ if it satisfies following characteristics:

(i) every label in $L(Q)$ is contained in a node N_p, i.e., the labels are equal,

(ii) all pairs of nodes (n_1, n_2) with $n_1, n_2 \in N_p$ are meaningful connected. Two nodes are meaningful connected, if

a) n_1 (n_2) is a general ancestor of n_2 (n_1), or

b) n_1 and n_2 have a common, general descendant element

(iii) there does not exist a meaningful schema pattern $P' = (N_{P'}, E_{P'})$ with $N_{P'}$ being a strict subset of N_P, and

(iv) for each node $n \in N_P$ and each edge $e \in E_P$: if n or e is removed, P does not satisfy condition (i), or P is no longer a connected graph.

The definition ensures that the schema pattern is total and minimal. The point (ii) describes meaningful connection between elements. The condition ii(a) is explained as follows. Assume the labels "museum" and "drawing". They have the shared common descendant "location". That means, a drawing is exhibited in a museum. Thus, the data items with label museum can be connected to data items with label drawing. Based on the meaningful schema patterns, Yu and Jagadish define a subtree of the data graph D as meaningful result with respect to Q if

(i) the subtree conforms to a meaningful schema pattern with respect to $L(Q)$ and

(ii) for every labeled keyword $l : kw \in Q$, the subtree contains a node n with n contains kw and has the label l.

General graph. The previous approaches deal with XML data trees or are schema-based graphs. There are keyword search systems on a directed data graph $G_D = (V, E)$, too. The idea is now to find subtrees or sub-graphs in this graph that contain all keywords. For example, a relational database can be modeled as directed graph of tuples [HN02]. Web page graphs are another example. Finally, one can model heterogeneous data (relational database, XML, unstructured documents) in one data graph [LOF$^+$08]. Possible meaningful results are classified into tree-based and sub-graph semantics [YQC10]. A **subtree** t of a data graph is a valid answer if all leaves[3] contain at least one keyword of the query Q, all keywords are in t, and no node can be removed without breaking the other conditions. There are two different tree semantics: **Steiner tree semantics** and **distinct root semantics**. Assuming the edges in the graph have weights, steiner trees are those subtrees with the lowest sum of edge weights. In contrast, subtrees with distinct root semantics are subtrees where every path of a root to the keyword containing node is minimal. Thus, for every node in the graph exists maximal one subtree with distinct root semantics [YQC10]. Further result types are **sub-graphs** that have a certain size.

The overview shows that the definition of a meaningful result in a structural database is not a trivial task. The result definitions are simple lowest common ancestor nodes without considering structure, trees, trees with special characteristics (label names), and trees according to a schema summarization. It can be seen, that it is necessary to include more semantics into the result construction in order to avoid too many and unnecessary, not meaningful results.

4.2.3. Ranking Methods

An appropriate definition of meaningful results avoids the construction of large amounts of redundant results. However, approximate queries still produce large sets of results that have to be ranked according to their relevance to the query. In classical

[3] All nodes that have only one or zero connections to other nodes.

information retrieval [BYRN99], documents are ranked using statistics about terms and documents in a document collection. Integrating keyword search into structural data sources extends the problem. Firstly, it has to be defined, which statistics are necessary for the search and how are they obtained. Secondly, queries in structural sources may consist of approximate structural and keyword query parts. That means, the structure has to be considered in the score, too. In this section, we classify scoring approaches in the areas of content scoring, which describe different approaches of keyword-based scoring of nodes and structural scoring that determines the relevance of a result to an approximate structural query. Finally, we discuss methods to combine both, keyword score and structural score.

Content scoring

Keyword scoring describes the relevance of a result with respect to a keyword query. Similar to information retrieval on flat documents, different approaches exist:

- Boolean retrieval [FKM00]: the relevance of a result element is determined, if the element contains all conjunctively connected keywords, or any of the disjunctive keywords.

- Vector space model [CMKS03, TW02a, CEL+02]: the vector space model assigns weights to terms in queries and result elements [SM83]. Both, queries and result content, are represented as $|\mathcal{T}|$-dimensional vectors of term weights. The similarity between results and query are computed using the cosine measure, which is defined as

$$\begin{aligned} sim(n_j, q) &= \frac{\vec{n_j} \bullet \vec{q}}{|\vec{n_j}| * |\vec{q}|} \\ &= \frac{\sum_{i=1}^{|\mathcal{T}|} w_{i,j} * w_{i,q}}{\sqrt{\sum_{i=1}^{|\mathcal{T}|} w_{i,j}^2 * \sum_{i=1}^{|\mathcal{T}|} w_{i,q}^2}} \end{aligned} \tag{4.1}$$

  with n_j and $\vec{n_j}$ being a node and the corresponding vector representation, q and $\vec{q}$ a query and its vector representation, respectively, and $w_{i,x}$ the term weights in node n_j $(x = j)$ or query q $(x = q)$, for $1 \leq i \leq |\mathcal{T}|$. Zobel and Moffat classify related similarity measures [ZM98]. Singhal presents another, highly optimized variant [Sin01].

- Probabilistic information retrieval model [Fuh92]: using the probabilistic information retrieval model we aim to estimate the probability that a specific result element will be judged relevant with respect to a specific query. Using a set of terms representation of queries and documents, assuming independence of terms in the document, and removing constant equation terms, elements are ranked using Equation (4.2)

$$score(N_n, Q) = \sum_{t_i \in Q \cap N_n} \log \frac{p_i * (1 - q_i)}{q_i * (1 - p_i)}, \tag{4.2}$$

where Q and N_n are two term sets representing the query and the node, p_i the probability $P(t_i|N_n)$ that t_i is contained in N_n and q_i the probability $P(t_i|\overline{N_n})$ that t_i is not contained in N_n. The probabilities may be estimated using term weight statistics [Sin01]. Approaches using the probabilistic information retrieval model are the query languages XIRQL [FG04] and XXL [TW01].

Basically, all content scoring models rely on some kind of term statistics. The most common and relevant statistics are the term frequency within a result r, denoted as $tf(t, r)$, which describes the number of occurrences of term $t \in \mathcal{T}$ in result r, the document frequency $rf(t)$, that describes the number of results containing term t, and finally, the normalization factor $rlen(r)$, which denotes the length of the text in a result. In the case of information retrieval in structured databases, the question arises, how to define the statistics [AYL06].

At first, there are different approaches to compute $rf(r)$. The result frequency may incorporate all nodes [CMM+03], all nodes of the same type [TW02a], or only leaf nodes that contain the actual text [CMKS03]. The statistics of the leaf nodes are translated to internal nodes taking the distance into account. Computing the document frequency rf based on the selected search context is a dynamic ranking approach [BG06, BS05]. The idea is that the importance of a term in a subset of results, possibly defined by a structural constraint, differs from the importance in the complete database.

The values $tf(t, r)$ and $rlen(r)$ may be computed by concatenating the texts of the containing nodes if the result r consists of different textual nodes. Other approaches obtain statistics and similarity values for each leaf node separately and subsequently, combine the similarity values [HGP03]. Liu et al. [LYMC06] discuss the effective term weight computation for information retrieval in relational databases. We discuss their approach further below in this section.

Using the boolean retrieval model, a ranking of result elements may be based on the element rank [HN02, GSBS03] inspired by the PageRank algorithm [BP98, PBMW98]. The score of the result element depends on the importance and the distance to the actual content nodes as well as the size of the result. Here, we find a bridge to structural scores that are described in the following.

Structural scoring

The structural score represents the closeness of the structural result with respect to an approximate structural constraint. Furthermore, it complements keyword scores by taking the structure of results into account, even for structure-free queries.

Size and distance of the result. A result of a pure keyword query is either a node, which is the lowest common ancestor of content nodes or a set of connected nodes. The size of the result as well as the distances have to be considered to improve the ranking. Generally, smaller results, e.g., results comprising fewer nodes, or more specific results, e.g., the distance to content nodes is smaller, are ranked higher [HP02, ACD02, GSBS03].

A first basic structural score is the (inverse) size of the result. The size is defined as the number of all nodes or number of node connections in the result [ACD02, HP02, CMKS03]. A more detailed approach [HN02] uses edge weights between result nodes and the closeness of the corresponding schema nodes, e.g., relations. Edge weights may represent different relationships between nodes. For instance, link edges may have a smaller value than containment edges.

A second basic structural information is the distance between result node and content nodes. Proximity search in data graphs selects directly nodes that are as close as possible to another set of nodes [GSVGM98]. The content score is decreased with increasing distance between answer nodes and content nodes [GSBS03].

Node name similarity. A first relaxation of structural constraints is the expansion of node labels [TW02a, LYJ04, FG04]. Using element name expansion the result is higher ranked if the element names are more similar to the query. Different approaches exist to describe the similarity between node names. Most often a distance measure is used. A common approach is the Levenshtein or edit distance [Lev66], which describes the distance between two strings using the number of edit operations that are necessary to transform one string into the second.

Approaches using the edit distance cannot handle situations with semantically similar element names, which are syntactically different. Synonym word lists, vocabularies, or an ontology have to be used in order to expand query node labels. One approach to describe the accurateness of a name with respect to a query name is the distance between the two names in an ontology graph [TW02a], the closer the names, the higher results are ranked. Furthermore, user-defined functions may be used [FG04].

If several node labels have been expanded, the similarity result are considered as probabilities and be combined using probabilistic functions [TW02a, FG04].

Scoring of query relaxation. Node name expansions do not allow the relaxation of structural constraints like edge generalization, subtree promotion, and leaf node deletion. Each of these relaxations has to be scored according to the following rules [AYLP04]:

1. relevance scoring: the structural score of a relaxed query is lower than the score of a less relaxed query,
2. order invariance: the relevance score is independent of the order of applied relaxation operators, and
3. efficiency: the score has to be computed efficiently.

Generally, result elements belonging to a result set of a less relaxed query have a higher score than results of a more relaxed query. The task of scoring query relaxation is to determine the influence of a relaxation operation to the result.

A first approach uses *penalties* for each relaxation [AYLP04]. The penalty is proportional to the quotient between the number of the results of the original query and the relaxed query. Consequently, the score is the sum of all query predicate weights decreased by the sum of penalties.

A second approach [AYKM+05] is based on $tf.idf$ values. Here, The idf value describes the quotient between the result size of the least relaxed query and the result size of the current relaxation with respect to a given database. According to that, the idf value of a result element is the maximum idf of a query relaxation (most specific relaxation) whose result set contains the element. The result nodes are roots of document fragments that satisfy a structural pattern. The document fragment is denoted as match. Every result node may be the root of different matches. The tf value of a result element corresponds to the number of different matching data trees with one common result node as root. The complete structural score combines idf value and tf lexicographically. That means, the score of node n' is smaller than the score of n if

- $idf(n') < idf(n)$ or
- $idf(n') = idf(n)$ and $tf(n') < tf(n)$.

Another approach [Sch02] describes the score as the costs that are necessary to transform a query into a relaxed query. The transformation is defined by a sequence of basic transformations: insertion of nodes, deletion of inner nodes and leaf nodes as well as label renaming. Every basic transformation has a cost value, which is assigned by an administrator. In consequence, the score of a relaxed query is the sum of the costs of all basic transformations.

JuruXML [CMM+03] uses XML fragments as queries which comprise approximate XML structure and keywords. The context is given as a path of element names, e.g., $q_1/q_2/q_3$. Results are ranked using an extended vector space model that combines keyword and structural scores. The structural score is computed by a function cr which maps the context resemblance to a real value in the interval $[0, 1]$, i.e., the structural closeness of the answer according to the query. The function cr includes the following ideas [CEL+02]: cr will return a high value, if

- the context of the result satisfies many constraints (element names) of the query in the right order,
- the result context matches the query context at the beginning of a path,
- matching path information are close together, i.e., if matching element names are in the path close together and not spread over long path, and
- the path of the answer approximately has the same length as the query path.

Combining structural and content Scoring

Content scores only are not sufficient to generate a meaningful ranking [AYL06], a *combination of evidence* is necessary, thus, structural scores and keyword scores have to be combined. Following basic approaches exist [AYLP04]:

1. structural score first,
2. keyword score first, or

3. a combination of both approaches

Next, we discuss several combination approaches. Keyword search in relational databases finds tuple trees that satisfy a keyword query. These results are ranked by a score function which combines the size of the tree and the keyword score. Hristidis et al. [HGP03] generalized different approaches as

$$Score(T, Q) = Combine(\mathbf{Score}(A, Q), size(T)) \tag{4.3}$$

with T a tuple tree of the size $size(T)$, A a set of attributes in T, and $\mathbf{Score}(A, Q)$ a function which combines the keyword scores in each attribute a_i of A. Proximity search in structural databases utilizes only the distance between two result nodes to determine the rank [GSVGM98, DEGP98a, DEGP98b].

The study of Liu et al. [LYMC06] focuses on the effectiveness of IR-search in relational databases. As in information the term weights are most relevant to find good rankings [Sin01], Liu et al. provide effective term weights for tuple tree scoring. The weights are based on four normalizations: (*i*) tuple tree normalization,(*ii*) document length normalization, (*iii*) document frequency normalization, and (*iv*) inter-document weight normalization. Equation (4.4) describes the term weight of a term t in the document D_i; a document corresponds to an attribute value of one attribute:

$$w(t, D_i) = \frac{ntf * idf^g}{ndl * Nsize(T)}. \tag{4.4}$$

The components of the weight $w(t, D_i)$ are in particular: the normalized term frequency

$$ntf = 1 + \ln(1 + \ln(tf)),$$

which describes the number of occurrences of a term in the document; the global inverse document frequency

$$idf^g = \ln \frac{N^g}{df^g + 1}$$

that represents the document frequency normalization with N^g the total number of documents and df^g the global document frequency; the normalized document length

$$ndl = \left((1 - s) + s * \frac{dl}{avgdl}\right) * (1 + \ln(avgdl))$$

with dl the current document length and $avgdl$ the average document length in a one attribute, s a weighting factor; the normalized tree size of tree T is defined as

$$Nsize(T) = (1 - s) + s * \frac{size(T)}{avgsize},$$

with $avgsize$ the average size of all answer trees. The value $w(t, D_i)$ represents the weight of a term in one document, thus, the particular weights of all documents in a

result tree T have to be combined. Let $D_i, 1 \leq i \leq m$ be the documents in a tree T, then Equation (4.5) defines the combined term weight:

$$w(t, T) = \mathbf{Combine}(w(t, D_1), w(t, D_2), \ldots, w(t, D_m)). \tag{4.5}$$

Liu et al. [LYMC06] define the function **Combine** as

$$\mathbf{Combine}(w(t, D_1), w(t, D_2), \ldots, w(t, D_m)) = maxWgt * \left(1 + \ln\left(1 + \ln \frac{sumWgt}{maxWgt}\right)\right)$$

with $maxWgt = \max(w(t, D_1), w(t, D_2), \ldots, w(t, D_m))$ and $sumWgt = \sum_{i=1}^{i=m} w(t, D_i)$. The function **Combine** realizes the inter-document weight normalization. However, the definition of Liu et al. is not a monotonic function and requires specific algorithms to be used efficiently.

Further ranking approaches

The keyword score in XSEarch [CMKS03] is computed using the vector space model. The score is modified by division by the result size and multiplication of the weighted number of ancestor-descendant pairs to include structural information.

XRank [GSBS03] uses the importance of a content node and the distance of the result node to the content node as ranking of XML nodes. BANKS [HN02] utilizes a similar ranking for relational databases.

The XXL system assumes that all scores, keyword scores and similar element name scores are unweighted probabilities and combines the scores using probabilistic formulas for `AND` and `OR` operators [TW01]. XIRQL interprets all single ranking information as weighted probabilistic events, and provides operators to combine the values [FG04].

A different approach uses JuruXML [CMM+03]. This system searches for term context pairs and ranks the results using a vector space model. The weights for term context pairs are based on the common tf and *idf* values, but are extended by context resemblance values between query and result context. In order to avoid high *idf* values for rare contexts, several *idf* merge approaches were proposed. That means, JuruXML modifies the term weights, but not the final similarity values.

The works [BG06] and [BS05] include the contexts of query and answers into the final score by computing tf and *idf* values dynamically depending on the current context of the XML document and query.

Botev et al. [BAYS06] and Al Khalifa et al. [AKYJ03] present two general languages for full-text operations on XML data and structural databases. The former approach is based on relations of the token positions. Using these relations and a scoring function, operators are defined, which create and modify result scores. The latter approach is based on scored data and pattern trees. Scored pattern trees represent queries, which comprise a result structure (node names, parent-child, or ancestor-descendant relationships) and keyword queries. Furthermore, user specified scoring functions are used on each pattern tree node. Scored data trees are subtrees of the data graph. Nodes matching a keyword query are attached with a score. Internal node scores are

computed using user defined score functions and scores of child nodes. The final score of a scored data tree is the score value of the root node or the highest score value in the tree.

4.2.4. Query Processing in Keyword Search Systems

There are two basic approaches to search for results in structural databases (see Figure 4.3). The first approach extracts first candidate networks from a schema graph and retrieves matching results, subsequently. The second approach works directly on the data graph or an index representation of the data graph to construct results.

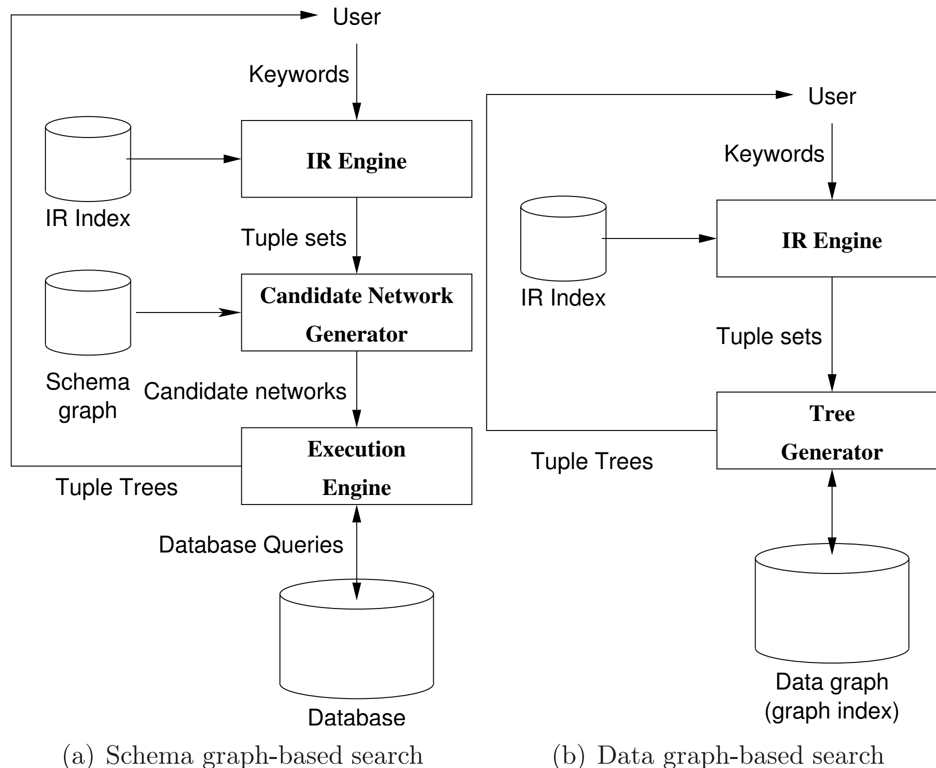


Figure 4.3.: Schema graph-based vs. data graph-based keyword search

Schema graph-based Search

Schema-graph based query processing comprises following steps (Figure 4.3(a)):

1. extracting schema nodes matching the search condition (e.g., labels) or supporting search keywords using a master IR (information retrieval) index,
2. creating result candidates from the schema nodes using the schema graph,
3. create query patterns using candidates and keyword containment conditions, and
4. execute queries using the query patterns and obtain the results.

The first step depends on the kind of query. If the query has been a labeled keyword query, the schema nodes are selected based on the label set of the query [YJ07]. In the case of keyword-only queries, a master IR index is used that relates keywords to containing attributes [ACD02]. Additionally, the algorithm of [LYMC06] tries to find *schema keywords* from a keyword query, which relate to an attribute or relation name. The decision is based on possible scores of results, i.e., term weights. The result of the first step is a set of schema nodes and corresponding lists of query keywords or candidate node sets associated to the schema nodes.

In the second step, candidate networks are created using the schema graph and the result of step one. Candidate networks are subgraphs of the schema graph that contain at the leaves schema nodes with an associated set of keywords. Inner nodes may have empty keyword sets. The candidate network has to contain all query keywords (AND semantics) or at least some keywords (OR semantics). Furthermore, it is required that a network is minimal.

Based on these candidates, query patterns are constructed. That are either pattern trees for XML databases [YJ07, AKYJ03] or join trees for relational databases [ACD02, HPB03, HGP03, LYMC06]. Query patterns include the structure of the schema network and the keyword conditions using the associated keywords. The last step is the execution of query patterns as well as ranking and merging of the obtained results.

Data graph-based search

Given are a query and a data graph or an index representation of the graph in main memory. The results are constructed in the following steps (Figure 4.3(b)),

1. nodes containing any keyword of the query are retrieved using a master IR index,

2. the obtained nodes are combined to results using the data graph or specialized merging operations.

We do a lookup in the IR index in the first step. The master index contains for each keyword its occurrences, i.e., the containing nodes. The result of the first step is a node list for each keyword of the query. In the second step, the node lists are combined to results. Here, the system utilizes the data graph or an index representation to create valid sub-trees or subgraphs. Systems use specialized graph algorithms to construct results [HN02, GSVGM98, DEGP98a, LOF^{+}08], for instance Steiner tree search [Ple81] or shortest path algorithms, e.g., Dijkstra's algorithm. The data graph may be held in main memory [HN02] or interconnection indexes, e.g., Hub indexes, are utilized [GSVGM98]. In XML trees, stack-based algorithms in combination with corresponding index structures are used to compute lowest-common ancestors, e.g., [GSBS03, CMKS03, XP08].

Data graph approaches have the disadvantage of using a large graph in main memory that have to be maintained additionally to disk-resident inverted indexes. Furthermore, schema information is lost by unlabeled edges.

4.3. Keyword Search Systems

After describing the general concepts of keyword search in structured databases, we present several actual approaches in this research area. First, we discuss systems for keyword search in relational databases. Second, approaches for relaxed structural queries are described. Third, plain keyword search in XML as well as query language extensions for XML are discussed.

4.3.1. Relational Database Search

Relational keyword search systems use either the schema graph-based or data graph-based approaches. The results of the search systems are minimal joining tuple trees or sub-trees in data graphs. The results are ordered by different scoring functions. First, we describe for each approach an exemplary set of works. Second, we mention further works that improved these approaches or provided new approaches.

DISCOVER system [HP02, HGP03]

Hristidis et al. developed the DISCOVER system allowing plain keyword queries with AND semantics [HP02]. The system is a representative of the schema graph-based approach. It proposes a query optimizer in order to avoid redundant execution of query patterns or parts of query patterns, respectively. An Oracle text index implements the master index. The system ranks the results according to the inverse tree size. A similar approach is the DBXplorer system proposed by Agrawal et al. [ACD02]. The authors concentrated on an efficient master index structure for which different variants were proposed. Furthermore, DBXplorer uses a window function [Ull90] as network generation module.

The DISCOVER system was extended in order to allow OR semantics for queries and to use IR-scores in the ranking function [HGP03]. The ranking function combines scores of the query in every text attribute $\mathbf{score}(a_i, Q)$ with a_i a text attribute and the size of the tuple tree in the following way:

$$\mathbf{score}(T, Q) = \frac{\sum_{a_i \in A} \mathbf{score}(a_i, Q)}{\mathbf{size}(T)}.$$

The function satisfies the monotonicity condition. While the work of Hristidis et al. focuses on efficiency of the keyword search system, Liu et al. deal with effectiveness of tuple tree retrieval [LYMC06]. The authors extend the approach of [HGP03] by improved term weight normalizations for tuple trees. The resulting weights have been shown in Equation (4.4) and Equation (4.5). Furthermore, Liu et al. evolve the work in that way, that also schema elements are considered as possible keyword occurrence positions.

BANKS system [HBN+01, HN02, ACD+03]

The BANKS database search and browsing system was developed by Hulgeri et al. [HBN+01, HN02, ACD+03] and is a representative of data graph based query

evaluation. Results in BANKS are constructed as follows. A data graph $DG = (N, E)$ is constructed in main memory. Nodes are tuple identifiers and edges represent foreign key relationships between tuples. Nodes and edges are decorated with corresponding weights. Given a keyword query $Q = \{kw_1, kw_2, \ldots, kw_m\}$, in the first step a tuple set S_i for each keyword $kw_i, i = 1, 2, \ldots, m$ is retrieved using a disk resident, inverted index. The second step generates the tuple trees using the main memory data graph and the candidate tuples sets S_i in the following way. The BANKS system creates for every node $n \in \bigcup_{1 \leq i \leq m} S_i$ an iterator that implements Dijkstra's single source shortest path algorithm with n as source. If a node $n' \in N$ has been visited by at least one iterator of each keyword set S_i, it is the root (called information node) of a tuple tree. In that way, the system can find Steiner trees [Ple81] in a data graph. The tuple tree is ranked by a function including tuple weights and edge weights in the tree:

$$\mathbf{score}(T, Q) = \begin{cases} \omega \cdot Nscore(T) + (1 - \omega) \cdot Escore(T) & T \text{ contains all keywords} \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, the BANKS system allows the browsing of the results using a graphical interface [ACD+03]. Other data graph-based approaches are the database proximity search [GSVGM98] and the DTL Dataspot system [DEGP98a, DEGP98b].

The BANKS system requires the data graph to be held in main memory. Goldman et al. [GSVGM98] described how graph distances can be pre-computed and stored in secondary memory using hub-indexes. However, their approach only supports a less general kind of keyword query of the form *FIND x NEAR y*.

Further approaches

Both sets of works have been extended and improved by many other researchers and works. For example, the SPARK system uses a non-monotonic scoring function [LLWZ07, Luo09]. Therefore, the approach of Hristidis et al. cannot directly be used. A skyline processing of candidate networks is the solution. Other works improved the generation of candidate networks [MYP09, QYC11] by ordering relation names or using templates, respectively. Since during the schema graph-based processing many join queries are generated, large, intermediate result sets may be generated. Qin et al. proposed a two step approach to avoid these intermediate results by using semi-joins [QYC09]. In a first step, only join partners are selected from relations using a semi-join. In the second step, the actual joins are performed. Xu et al. ranked candidate networks first and modified the score of tuples [XIG09]. A candidate network has a high score if the relations are more typical for the keyword query. A relation is more typical with respect to a keyword, if it contains many tuples that contain the keyword and only few other relations contain the keyword, too. This definition follows the tf.idf paradigm (see Section 4.2.3).

Kacholia et al. extended the BANKS approach by a bidirectional search [KPC+05]. BANKS searches backward from the keyword nodes to find a tree. Kacholia et al. also allow the forward search to connect keyword nodes if a keyword node is contained by many tuples. Li et al. extended the data graph approach to heterogeneous data sources comprising relational databases, semi-structured, and unstructured

sources [LOF+08]. The authors also provide sub-graphs as answers instead of sub-trees and their efficient computation. Additional answer types for data graph approaches are discussed [LFZW11].

Besides schema and data graph approaches, other ideas were proposed. Masermann and Vossen [MV00] proposed the keyword search in databases using reflective SQL. Here, queries are treated as data and can be modified and executed subsequently. If a Boolean keyword is given, queries are modified in that way, that relevant tuples can be retrieved from *one* relation. That means, the approach supports keyword search for tuples of one relation.

Su and Widom [SW03, SW05] proposed the EKSO system that uses pre-computed tuple trees. In the pre-processing phase, the system retrieves all tuple trees from the database. First, a set of root relations were defined containing root tuples. *Text objects* are created following primary key to foreign key links. A text object corresponding to root tuple t consists of all tuples that join to t directly or indirectly. In a second step, a *Virtual Document* is created from each text object. Virtual documents are the concatenation of all text attributes of the text object and serve as input of a classic IR indexer. Keyword search is supported by an IR engine using the inverted index over the virtual documents. The result is a list of root tuple identifiers ranked by the relevance of the corresponding virtual documents to the query. The advantage is a time-saving pre-computation and reuse of existing IR engines. The disadvantages are a large index overhead and less flexibility in the answers according to size and structure of answer tuple trees. A second materialization approach is the RETUNE system [LFZ08]. The materialization element in RETUNE is a tuple unit. A tuple unit is a set of tuples that is created if we join a tuple of relation $r(R_i)$ with all connected relations $r(R_j)$. Now, for every tuple in the database the tuple units are created. The result of a keyword search is a tuple unit that contains all keywords of a keyword query.

Markovetz et al. [MYP09] and Qin et al. [QYC11] investigated keyword search over relational data streams. There are two challenges: efficient generation of candidate networks for a complete database schema in schema graph approach and the sharing of computations of many join operations during processing of many streams.

The last group of systems uses keywords to generate queries or forms instead of tuple trees [TL08, DZN10, CBC+09, BRL+10, BRDN10]. The user gives a keyword query, and the system generates possible queries of forms. Using them, the user executes the query.

4.3.2. Keyword Search in XML Trees and Graphs

At first, we want to discuss *structure-free* keyword search approaches. An early work is the boolean keyword search integration that is discussed by Florescu et al. [FKM00]. In particular, the authors discussed an inverted index, which uses a relational database system. In the following, we discuss keyword search approach over XML trees and graphs, query relaxation, and keyword search integration into XML query languages.

XRank [GSBS03]

The system XRank [GSBS03] accepts keyword queries of the form $Q = \{kw_1, kw_2, \ldots, kw_n\}$ and returns XML elements using exclusive lowest common ancestor (elca) semantics (Section 4.2.2 and [XP08]). The system ranks the results using a scoring function that includes the concepts: result specificity, keyword proximity, and hyperlink awareness. The result ranking formula is specified as

$$\mathbf{score}(n, Q) = \left(\sum_{1 \leq i \leq n} \hat{r}(n, kw_i) \right) \cdot \mathbf{p}(n, kw_1, kw_2, \ldots, kw_n).$$

The value $\hat{r}(n, kw_i)$ represents the element rank of n, denoted as $ElemRank(n)$, which is decreased with increasing distance between the node n and the node n' that contains the keyword kw_i. If the keyword kw_i occurs more than once in n, then the minimum distance will be used. The function $\mathbf{p}(n, kw_1, kw_2, \ldots, kw_n)$ models the keyword proximity, e.g., it computes the minimum window in which all keywords occur.

While the values $\hat{r}(n, kw_i)$ and $\mathbf{p}(n, kw_1, kw_2, \ldots, kw_n)$ meet the result specificity and keyword proximity requirements, the element rank $ElemRank(n)$ includes the third demand: hyperlink awareness. The element rank of a node n is computed using a PageRank [PBMW98] inspired algorithm. Here, the rank is specified by references via containment edges, inverse containment edges as well as link edges, and the rank of the referencing elements. The XRank system uses an inverted indexes where nodes are identified by Dewey numbers [Dew04] and containing elements are produced by stack-based algorithms.

XSEarch [CMKS03]

The XSEarch system looks for interconnected node sets (Section 4.2.2) that satisfy the query. The system accepts labeled keyword queries, i.e., queries of the form $Q = \{t_1, t_2, \ldots, t_m\}$ with $t_i = (l_i, kw_i)$ a pair of label $l_i \in Name$ and $kw_i \in \mathcal{T}$ a keyword. Given a query Q, a sequence of nodes and null values $R_Q = n_1, n_2, \ldots, n_m$ is an answer if the nodes are meaningfully related.

XSEarch ranks node sets using a modified vector-space approach. The keyword weights are computed using a $tf \cdot idf$ function, labels have a user-defined or system generated weight. Every non-leaf node has an assigned vector that has a dimensionality of $|Name \times \mathcal{T}|$. The answer R_Q is scored according to the following function:

$$\mathbf{score}(Q, R_Q) = \frac{\mathbf{sim}(Q, R_Q)^{\alpha}}{\mathbf{tsize}(R_Q)^{\beta}} \cdot (1 + \gamma \cdot \mathbf{anc\text{-}des}(R_Q))$$

with $\mathbf{sim}(Q, R_Q)$ the sum of the cosine measures between query and node vectors, $\mathbf{tsize}(R_Q)$ the size of the result, that means the number of distinct nodes, and $\mathbf{anc\text{-}des}(R_Q)$ the number of node pairs that are in ancestor-descendant relationship, which is assumed to be a close connection. The values α, β, γ are weighting factors. For fast result computation, the XSEarch system uses a specialized interconnection index.

XKeyword [HPB03, BHK+03, HKPS06]

Hristidis et al. extended the DISCOVER approach (see Section 4.3.1) for proximity search in semi-structured databases. This approach belongs also to the set of schema graph-based search approaches. In detail, the proposed system accepts plain keyword queries of the form $Q = \{kw_1, kw_2, \ldots, kw_n\}$ and the result consists of minimal connecting trees of target objects that contain all keywords of Q and are ranked by their size. That means, the query has AND semantics and smaller trees are ranked higher. The work mainly addresses two problems: the definition of meaningful results and presentation of a possibly large set of results. The proposed solution to the first problem is the introduction of target objects which are XML fragments. XML fragments are described by schema graph splits. These schema graph splits are denoted as target schema segments and describe minimal self-contained information pieces (see also Section 4.2.2). They are defined by the administrator.

The second problem, huge sets of results, is addressed by the approach of *presentation graphs*. A presentation graph hides structurally similar results, i.e., multi-valued dependencies between nodes. The XKeyword implements this approach [BHK+03].

Xu and Papakonstantinou [XP05, XP08]

Xu and Papakonstantinou propose several algorithms for smallest (slca) [XP05] and exclusive lca (elca) [XP08] results of keyword queries in XML trees. Given k sorted lists $S_1, S_2, \ldots, S_k$ of nodes for a keyword query $kw_1, kw_2, \ldots, kw_k$ they propose several algorithms for computing $scla(S_1, \ldots, S_k)$. The algorithms are based Dewey identifier operations and ordering of nodes. Given a node v and a list S_2, the authors show that they need only the largest smaller node[4] of v in S_2 and the smallest larger node of v in S_2. The smaller of those matches is an slca result. Thereby only the smallest list S_1 needs to be entirely scanned . This computation is efficient because of the use of an index. This approach was adapted to elca semantics [XP08]. The main contribution is the efficient computation of results in XML trees and not to provide the best ranking function.

4.3.3. Relaxed Structural Queries

Relaxed structural queries are a second part of flexible queries on semi-structured databases. The problem here is that a system has to use the structural constraints given by a user, but must not penalize the user for giving them.

Kanza et al. [CKS01, KS01, KNS02]

Kanza et al. deal with flexible queries over semi-structured data, e.g., XML data. The data is modeled as a rooted data graph. Queries are rooted graphs with labeled edges and nodes represent variables. The goal is to find matches that map the query graph to the data graph. A mapping will be valid if it satisfies one or more of the following constraints:

[4]Based on a pre-order numbering using Dewey identifiers.

- root constraint: the root variable of the query matches the root node of the data graph,
- edge constraint: an edge in the query has to be mapped to an edge in the data graph,
- weak edge constraint: the label of the incoming edge in a query matches the incoming edge label of the corresponding data node,
- quasi edge constraint: an edge in the query graph is matched by a path in the data graph.

Furthermore, the filter constraint is applied, if selections occur in the query graph. A rigid matching satisfies the root constraint as well as all edge constraints. That is the usual non-flexible query semantics. *Semi-flexible matching* is a matching between a query graph and a data graph which satisfies *(i)* the root constraint, and *(ii)* for each directed path in the query, there is a path in the data graph that contains the same labels. The labels do not have to be in the same order and do not have to be contiguous. Furthermore, *(iii)* a strongly connected component[5] in the query graph has to be mapped to a strongly connected component in the data graph to cope with cycles in the query and the data graph. A *flexible matching* of the query to a data graph satisfies *(i)* the root constraint as well as *(ii)* all weak and quasi edge constraints.

Amer-Yahia et al. [AYCS02, AYLP04, AYKM+05]

Amer-Yahia et al. investigate the relaxation of *tree pattern queries* or twig queries. A tree pattern query is a pair (T, F) where T is a rooted tree and F is a Boolean combination of value-based predicates [AYLP04]. Every node in $T = (N, E)$ represents a variable and the edges are labeled as *pc* for parent-child relationships or *ad* for ancestor-descendant relationship. The Boolean condition F contains for each variable a condition $\$i.tag = name$ which describes the label. The root node is the distinguished answer node. The result of a tree pattern query is defined as follows. All matchings to the query are obtained, i.e., sub-trees of the data graph that satisfy the structural conditions of the query. Given the matches, the final result set is the set of the root nodes of the matches. Note, that one root node may belong to different matches.

The following query relaxations are assumed in this context: edge generalization, subtree promotions, and leaf node deletion. Additionally, the operations node generalization [AYCS02] and *contains*-predicate promotion [AYLP04] have been discussed.

Example 4.3 *Figure 4.4 shows several tree pattern queries. (1) represents the original query. (2) represents a single query relaxation by generalizing the edge between artist and name to an ancestor-descendant relationship. Query (3) can be constructed by successive subtree promotions and leaf node deletions. Finally, query (4) represents a query which searches for a* `painting` *that contains the keywords "gogh" and "paris".*

[5] For each pair of nodes exists a directed path in a strongly connected component of a directed graph.

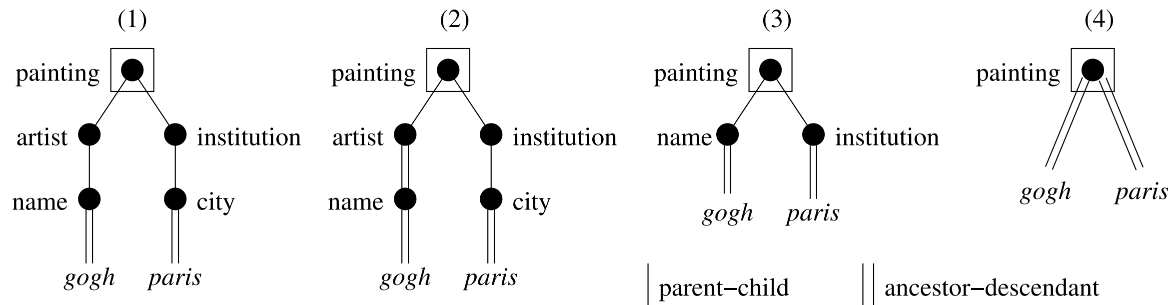


Figure 4.4.: Structural relaxation

The following contributions were made: given weighted query trees, an efficient top-k evaluation algorithm was proposed by encoding the relaxations into one query evaluation plan. Second, efficient algorithms for the combination of relaxed queries with keyword search have been shown using a structural answer score based on penalties for relaxations. At last, Amer-Yahia et al. introduce a similarity measure following the $tf.idf$ approach (see also Section 4.2.3). As twig scoring is expensive, approximate scoring mechanisms using path scoring have been developed.

Further approaches

Delobel et al. use tree query pattern relaxation within the semantic integration system Xyleme [DRR+03]. The system exhibits map translation tables that comprise the possible queries for a global tree query. As local data sources are mapped against a given global schema, some query parts may be empty. The following relaxation operations were introduced to overcome this limitation: unfolding tree nodes, deletion of conditions, and promotion of conditions. Unfolding tree nodes denotes the operation of duplicating a node in order to single out a query branch. In that way, join operations are moved to more general nodes.

Likewise, Schlieder [Sch02] investigates the relaxation of tree query pattern using tree edit operations. That means, a tree query pattern is modified using edit operations, and matching data trees are retrieved from the data ranked by the cost of the edit operations. The following tree edit operations, each assigned with separate costs, are considered: insertion of a node, deletion of inner nodes and leaves as well as renaming of node labels. A schema-based algorithm was proposed in order to retrieve the top-k results, i.e., root nodes of data trees that match to modified query trees with smallest edit operation costs. The algorithm is based on the ideas: encoding all edit operations except node insertion into the query tree, matching the query against a schema tree (replacing parent-child edges with ancestor-descendant edges for allowing node insertions), and retrieving matching sub-trees, which are used as queries to the actual data graph. An incremental top-k algorithm is used that retrieves first the best n modified queries and tries to find the required k results. As not all queries return a result, n has to be increased during query evaluation.

Several approximate XML languages [FG04, TW02a, CMM+03] also support structural relaxed queries. They are covered in the next section.

4.3.4. XML Query Languages and Keyword Search

In the recent years, many language proposals have been made for integration of keyword search into XML query languages [AYL06]. Mostly, new languages are based on existing structural XML query languages. The main principle of language integration is given in Figure 4.5 [BG02, AYBS04], which describes the interconnection of precise subqueries expressed in the host language and imprecise IR subqueries. First, precise

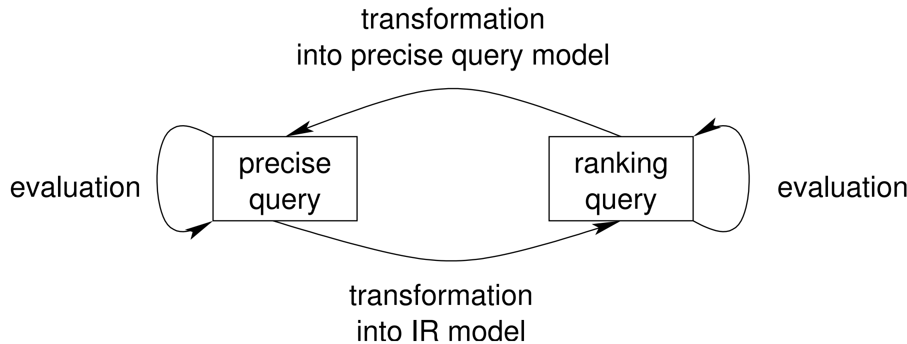


Figure 4.5.: Combination of precise querying and IR

subqueries are executed within the model of the host query language, e.g., XQuery. The results on the XML structure are expressed in the data model of the XML query language, e.g., sequences of items (elements and values). The results are transformed into the model of the IR subqueries, e.g., tokens and their positions in an XML graph. Based on this model, full text search operations can be expressed. Finally, the results are transformed into the XML query language model and the host language can post-process the results.

XIRQL [FG01, FG04]

The query language XIRQL is an extension of XQL and XPath, respectively, and has following features:

- index term weighting for query and document term and producing ranked results,
- retrieving most relevant document parts, i.e., a specificity-oriented search,
- datatypes with vague predicates, and
- structural vagueness to find close matches for structural query parts.

The query language introduces the operator `cw` for keyword search in an element. Search keywords are combined using different operators (AND, OR) and query weights can be assigned. The scores are combined using a probabilistic model. In order to support specificity-oriented search, index nodes with specific weights are introduced as result elements. Their weights are also embedded using probabilistic functions. XIRQL supports structural vagueness by generalizing attributes and elements, datatypes, and parent/child edges to ancestor-descendant edges (operator \\) as well as using similar element names that are specified in a vocabulary in the XML schema.

Example 4.4 *Assuming the query: Find paintings or similar which contain the phrase* ***blue horses*** *and* ***oil****, the resulting XIRQL query is illustrated in Figure 4.6.*

```
1 //cultural_assets\\~painting[
2                    .cw "blue horses" and .cw "oil"]
```

Figure 4.6.: Example query XIRQL

XXL [TW01, STW01, TW02a, TW02b]

XXL is a flexible XML search language that extends a part of the language XML-QL with the text similarity operator "∼". The operator can be used in unary form for element similarity operation and in binary form for attribute and element content comparisons. Similar element names are obtained from an ontology. The relevance of an element name is determined by the semantic distance in the ontology between query element name and answer element name. The relevance of content search keywords is computed using standard IR techniques as the cosine measure. The particular relevance scores are all in the range of $[0, 1]$ and are combined using probabilistic formulas. Specialized indexes for paths, ontology, and keyword positions are utilized for fast query evaluation [STW01]. Different query evaluation approaches have been investigated [TW02a].

Example 4.5 *For the query of the previous example: Figure 4.7 shows the resulting XXL query with "#" denoting an arbitrary path.*

```
1   SELECT P
2   FROM doc("art")
3   WHERE Cultural_Asset.#.(~painting) AS P
4         AND P ~ "blue horses" AND P ~ "oil"
```

Figure 4.7.: Example query XXL

TeXQuery [BSAY04, AYBS04] and XQuery Full-Text [AYBDS06]

XQuery is one of the most influential XML query languages [BCF+03]. TeXQuery and its successor, the W3C Query Full-Text, provide full-text search predicates for XQuery. The language extension consists of the new expression `FTContainsExpr` as well as score variables that are specified by the keyword `score`. The expression `FTContainsExpr` is defined as `FTContainsExpr := Expr ftcontains FTSelection` where `Expr` represents an XQuery expression, which defines the search context for the full-text selection predicate. XQuery Full-Text defines a set of fully composable full-text predicates comprising simple keyword selection, conjunctive and disjunctive

keyword connections, and phrase search as well as proximity predicates. The semantics of the expression and predicates are defined by the `AllMatches` data structure and operations on it which are defined as XML data types and XQuery functions, respectively.

The expression `FTContainsExpr` returns a Boolean value that indicates whether the full-text predicate is satisfied or not. Score variables are used to extract the score of a full-text expression. Score variables are indicated by the keyword `score` and are defined as float values in the range $[0, 1]$ where higher values denote higher relevance. Subsequently, the score variable can be used in an `order by` clause to sort the results by their score.

Example 4.6 *Using the exemplary query, the resulting XQuery Full-Text query is illustrated in Figure 4.8.*

```
1 for $p score $score in //painting[. ftcontains
2     "blue horses" && "oil"]
3 order by $score descending
4 return $p
```

Figure 4.8.: Example query XQuery Full-Text

Schema-free XQuery [LYJ04, LYJ08]

Schema-free XQuery is an extension to XQuery [BCF+03] supporting queries with only partial knowledge about the database schema. The language extension allows queries ranging from fully specified XQuery expressions to simple keyword queries. The results are specified as meaningful lowest common ancestor structures (MLCAS) as described in Section 4.2.2. At first, an XQuery function `mlcas` is introduced, which returns the root of an MLCAS for a given list of nodes. That means, the function `mlcas` finds meaningful relationships between the given nodes without requiring the user input. As `mlcas` returns an XML node, the results can be reused in other XQuery clauses. A further simplification is the introduction of the `mlcas` keyword[6]. The statements are rewritten into statements using the `mlcas` function. The following example shows the usage of the query extension.

Example 4.7 *Assume a query: Find* <u>*artists*</u> *and their* <u>*works*</u> *which belong to the* <u>*epoch*</u> ***renaissance****. The corresponding Schema-free XQuery expression is*

```
1 for $a in doc("art")//artists
2     $b in doc("art")//cultural_assets
3     $c in doc("art")//epoch
4 where $c/text() = "renaissance" and mlcas($a,$b,$c)
5 return <result> {$a,$b} </result>
```

The MLCAS are computed using a stack-based algorithm.

[6] In [LYJ08] the function is denoted as `mqf`, meaningful query focus, the new introduced keyword is denoted as `related`

Further approaches

Besides the previously mentioned systems several other approaches have been proposed, e.g., XIRCUS [MBHW02], ELIXIR [CK02], JuruXML [CEL+02, CMM+03], NEXI Content-And-Structure (CAS) [TS05], and XQuery/IR [BG02, BG06]. XQuery/IR extends XQuery by a `rank by` operator and provides mechanisms for context-based ranking. JuruXML uses as query language XML fragments, encodes query structure relaxation into term weights, and ranks the results based on the vector space model. The system DaNalix [LYJ07, LCY+07] provides a natural language interface to the user. The query expressed in natural language is analyzed and subsequently translated into Schema-free XML query statements. The work of Yu et al. [YJ06] summarizes the schema graph to important parts that the user can easily understand. Schema summarization is used to implement an XQuery extension which accepts labeled keyword queries and combines results without requiring structural knowledge by the user [YJ07]. The approach is denoted as Meaningful Summary Query.

Several researches deal with the semantics of structural keyword query languages [AKYJ03, Feg04, BAYS06, AYCD06]. Al-Khalifa et al. [AKYJ03] proposed query semantics based on a bulk-algebra TIX, which uses scored pattern trees with keyword conditions as queries and scored data trees as results. Botev et al. [BAYS06] and Amer-Yahia et al. [AYCD06] present languages and algebras for structural keyword search that are based on term position relations. A term position describes the location of occurrence of a term. Operations over term position relation allow the combination of different keyword relations. These approaches are similar to the `AllMatches` semantics of XQuery Full-Text [AYBDS06].

4.4. Keyword Search and Virtual Integrated Data Sources

In this section, we discuss keyword search over virtual integrated data sources. Here, we can distinguish the following classes of systems and approaches:

1. meta-search engines and distributed information retrieval over unstructured databases [Cal00],
2. structured Deep or Hidden Web [Ber01] searches, and
3. keyword search and approximate queries over structured sources in Intranets or relatively small closed domains.

Furthermore, the cases can be subdivided according to the behavior of the source: cooperative and uncooperative sources. The first two kinds of systems deal with relatively large numbers of sources which have to be selected according to a given query and the results have to be merged. The information is stored in different sources. In contrast, the last kind of systems deals with the problem of finding information spanning over different sources using keyword queries.

4.4.1. Metasearch Engines

Metasearch engines [MYL02] or distributed information retrieval systems [Cal00] are systems that provide a unified access to a number of local search engines and return merged top-k results to the user. The system deal with keyword search queries and work on unstructured results. Generally, query processing in metasearch engines comprises the following steps: after receiving the user query, e.g., a list of keywords, the *database selector* chooses the best databases with respect to a query, i.e., search engines with corresponding document collection. In the second step, for each selected database the *document selector* decides which documents have to be received. Here, the number of documents or the local similarity thresholds are specified. In the next step, the queries are translated into the format of the local search engines and are sent to them by the *query dispatcher*. The results obtained from the local sources are merged by the *result merger*. The result merger ranks the local results according to the global query and a similarity function and returns the best k result documents to the user. Figure 4.9 illustrates these components and steps. In summary, the main tasks of a metasearch engine on unstructured data are: database selection, document selection, and result merging.

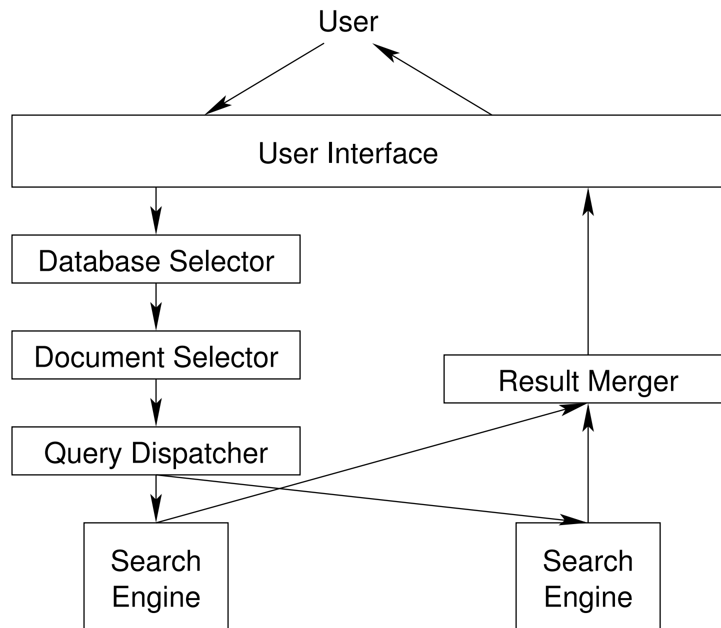


Figure 4.9.: Metasearch engine components [MYL02]

Database selection

The database selection problem considers two critical points: how are local database contents described? How to select the potentially most useful databases according to database descriptions and global user queries?

On the one hand, database descriptions have to represent heterogeneous content sufficiently to allow an effective database selection. On the other hand, descriptions

have to be easily maintainable to allow the scalability to large numbers of databases. Furthermore, creation of descriptions has to support cooperative and uncooperative sources [Cal00]. Three main approaches exist [MYL02]: rough manual descriptions, statistical content descriptions, and description learning by analyzing user feedbacks.

Rough manual descriptions contain high level information about a database, e.g., a short description of the database's subjects. While the descriptions are compact, they are inexact for heterogeneous sources and do not scale to high numbers of sources because of the manual creation.

Statistical representations contain summarized frequency statistics of a document collection. Analogously to single source statistics, database statistics consist of term statistics, denoted as *unigram language model* [Cal00]. The descriptions contain for each term different values, e.g., document frequency, aggregated term weights in one source. Additionally, the database size may be stored. In case of cooperative sources, the representative statistics are easily exchanged using standard protocols, for example, STARTS [GCGMP97] and SDARTS [GIG01, IBG02]. *Query-based sampling* [CCD99, CC01] is used to obtain statistics from uncooperative sources.

Query-based sampling is based on the hypothesis, that the language model obtained from a small sample of documents is similar to a complete resource description. As the sources do not allow a complete scan, traditional techniques as reservoir sampling [Vit85] cannot be used. Query-based sampling uses a number of random keyword queries to retrieve a sufficiently large set of documents, from which the resource description is created. Different studies showed the effectiveness of query-based sampling in distributed IR [CC01, SC03] as well as in database classification [IG02].

Finally, database description can be constructed using learning methods that analyze the user feedbacks about the quality of query results. Three kinds of learning methods are used [MYL02]: static learning using training queries, dynamic learning by analyzing user behavior after actual queries as well as combined approaches. Statistical descriptions are considered as the best and most flexible methods for the database selection problem [MYL02, Cal00].

Now, the database selection problem can be defined as choosing most potentially useful databases for a given information need and given resource descriptions. Most database selection approaches adapt document ranking techniques for this purpose in the following way. Every database is seen as one document. The term frequency is the number of documents in the database that contain the term. The set of all local databases is seen as collections of documents. Hence, the document frequency of a term is the number of databases that contain the term. In consequence, one can use the document ranking functions as database ranking functions. Popular approaches are the CORI approach [CLC95], which uses a Bayesian Inference Network, GlOSS [GGMT94] and g/GlOSS [GGM95] using the vector space model as well as approaches using additional semantic descriptions, e.g., [IG02].

Document selection

Document selection deals with the problem of how many documents have to be acquired from the local sources. It uses the local similarity function in order to maximize the effectiveness with respect to the global similarity measure and minimizing

the amount of retrieved documents. Meng et al. [MYL02] summarize the following approaches: *(i)* user-defined number of documents, i.e., the user specifies for each source a number of required documents; *(ii)* weighted allocation, i.e., the number of documents to retrieve depends on the rank of the selected database — a higher database rank implies more documents to select; *(iii)* a learning-based approach in which the number of documents is determined by past user experiences; *(iv)* guaranteed retrieval is a formal method to ensure that all potential documents are retrieved for any given query.

Result merging

Each searched database returns a sequence of ranked results, which have to be merged by the distributed IR system into one result list. Heterogeneous document ranking and score values cause the main problem of merging. That is caused by different corpus statistics, different algorithms, and different ranking and score representations. Solutions are [Cal00]:

- computation of normalized scores based on global statistics,
- recomputation of the relevance score on the global search client,
- estimation of normalized scores, or
- merging result lists using unnormalized scores.

The different methods require different levels of cooperation of the local sources, beginning with full access to the data ranging to different levels of ranking algorithms descriptions to uncooperative sources. Hence, the merging algorithm has to be chosen based on the given databases.

4.4.2. Structured Hidden-Web Search

The Deep Web consists of large numbers of unstructured and structured sources. The majority of these sources are structured databases [CHL+04], i.e., the databases have a query form consisting of different predicates and corresponding connections and deliver also structured results, e.g., lists of attribute-value-pairs. The following steps are required to meet the challenges of querying Deep Web sources [CHZ05, HMYW03]:

Find deep Web sources: First, the Deep Web sources have to be found, i.e., the start pages to or search form of the local databases. Specialized crawlers have to be developed, because manual indexing is not scalable.

Query interface extraction: Second, the query interfaces have to be extracted and to be described. The system has to find query interfaces, e.g., HTML forms, and has to interpret the possible predicates and their possible connections.

Schema Matching: Predicates consist of attribute names, comparison operator, and allowed values, i.e., the data type. It is necessary to translate queries from one source (or global layer) to other source queries. Hence, we need schema matching,

which relates an attribute to other attributes. As the number of sources is high and often changing, automatic schema matching techniques have to be used.

Source Clustering: Source clustering groups sources into categories that are organized hierarchically. Furthermore, a unified query interface is constructed for each category, e.g., a standard interface for book search. The interfaces are presented to the user.

Query Transformation: Query transformation is needed in order to translate queries against standard query interfaces to local query forms. The following types of heterogeneity have to be considered [ZHC05]: attribute level, predicate level, and query level. Attribute level heterogeneity denotes that two sources represent the same concept with different attribute names. If two sources use different predicates for the same concept, we speak of predicate level heterogeneity. For example, consider the search for items using a predefined price range. In different sources, the pre-defined ranges are often different. Query level heterogeneity comprises the differences in combinability of predicates in different sources.

Source Selection: Given a user query, relevant sources have to be selected. The sources can be selected based on query capabilities (e.g., supported attributes) or data quality.

Result Compilation: Result compilation consists of the extraction of results from Web pages, transformation into the global format as well as the combination of results across different systems.

As in the Hidden Web, similar to the flat Web, the sources are constantly changing, all sub-problems have to be solved automatically. Algorithms have to be general, but have to use domain-specific information like domain-specific ontologies or taxonomies [ZHC05].

The MetaQuerier system [CHZ05] integrates all these points as single subsystems. However, the integration does not only allow complete coverage of the Deep Web search, but also helps to improve the single systems by result feedbacks and ensemble techniques.

A second project for integration of Deep Web sources is the E-Metabase project [HMYW03, HMYW05]. A central part of that system is the WISE-Integrator that integrates search interfaces from different sites. Here, all information is used that can be extracted from HTML forms: attribute label, data types, value ranges, positions in the forms, domains (infinite, finite, range, hybrid), default values as well as positions in forms. Based on this information and semantic relationships (e.g., synonyms, hypernym, meronym) the schemas, i.e., the forms, are integrated. Furthermore, using clustering algorithms, representative attribute names are automatically constructed. Global interfaces are built automatically for each domain containing the most important attributes found in the local schemas.

The integrated global schemas introduce a new problem: the user has to work with complex, domain-dependent search interfaces. Initially, the right domain has to be chosen, secondly, the user has to get an overview over the offered query interfaces

and, thirdly, some query capabilities like disjunctive queries are hard to implement with normal query forms. Therefore, simplified keyword-based search interfaces were proposed.

Li et al. [LMM07] proposed a keyword-based search interface on top of the WISE-Integrator [HMYW03, HMYW05]. The system, denoted as EasyQuerier, provides a keyword-based search interface and maps the keywords to domains, e.g., book or job databases, and subsequently, to attributes and predicates. Mapping and query translation utilize metadata information from global and local interfaces as well as semantic dictionaries.

Calado et al. [CdSV+02, VCdS+02] proposed a system that structures keyword-based queries for structured Web databases in the Deep Web. In this case, the keyword queries are mapped directly to local query interfaces, i.e., attributes, instead of mapping to integrated query interfaces. In the first step, structured queries are created. The structured queries are ranked using a Bayesian network model and executed, i.e., sent to the sources. Necessary information are terms that occur in the database and the structure of the query interface. Uncooperative sources can be supported using query-based sampling techniques [CC01]. After execution of the queries, the result objects are ranked against the structured query and presented to the user. In both cases, EasyQuerier and the system of Calado et al., results that span different sources or domains are not supported. In the next section, we discuss this problem.

4.4.3. Virtual Integrated Structured Sources

Now we assume the scenario that information is spanning over virtually integrated, heterogeneous, structured data sources. That means, a result, e.g., a tuple tree or an XML node tree, contains parts from different sources. The described scenario emerges especially in cooperative Intranets of large companies or agencies or small closed domains of Web sources, e.g., Lost Art databases, which is exactly the working field of mediator systems (see Section 2.1.2). Thus, it is obvious to use existing mediator systems or existing integration techniques to extend single database keyword search solutions (see Section 4.3) to virtual integrated data sources.

Keyword Search across heterogeneous relational databases

In Section 4.3.1, we described the problem of keyword search in relational databases and corresponding solutions. Sayyadian et al. [SLDG07] extended that problem to the search across heterogeneous relational databases. The problem is defined as follows. Given a pure keyword query Q, the top-k minimal joining tuple tress (as defined in Section 4.2.2) in the integrated database are the result of Q. Following settings are considered in the given scenario: the local database systems are cooperative, but heterogeneous on schema and data level; the information need of the user is changing and on a short-term basis; foreign key relationships are exact within one database, but approximate across several sources. The system consists of two phases: the offline and the online phase (see Figure 4.10).

The offline preprocessing has two tasks: in the first task, the DBMS integrated IR indexers of the local database systems are used to build IR indexes for each database.

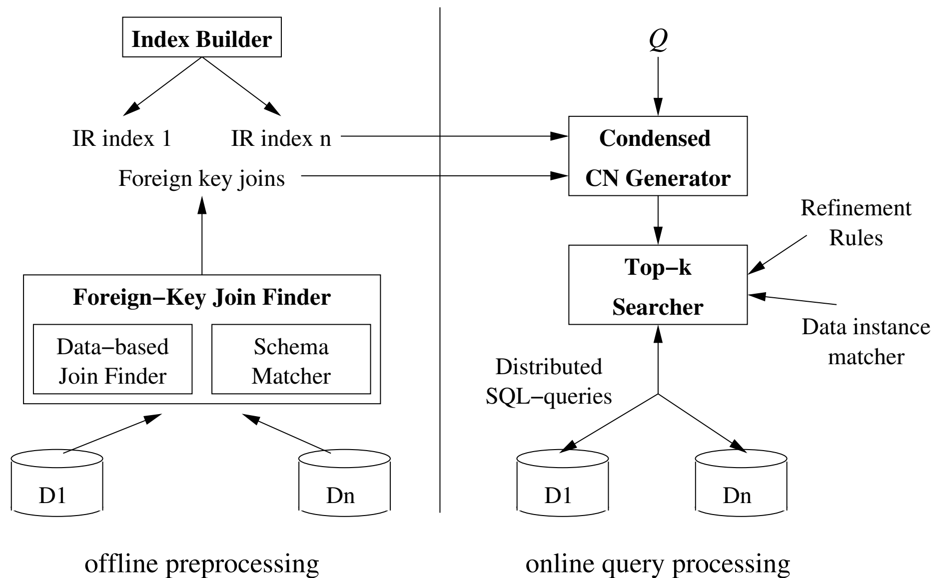


Figure 4.10.: KITE system architecture [SLDG07]

The indexes include keywords and their occurrences in tuples and attribute values. In the second task, foreign keys across different databases have to be discovered. The second task is implemented using the following steps. Firstly, approximate keys are searched for each table because given keys based on id attributes cannot help with inter-database joins. Secondly, the system tries to find matching join attributes to the constructed approximate key attributes. Thirdly, the foreign key joins are enumerated, i.e., for each pair of tables may exist several foreign key joins that comprise several key attributes and corresponding referencing attributes. The first three steps use techniques on the data level. The last step removes semantically incorrect joins by applying an automatic schema matching algorithm. That means, if attributes in a foreign key relationship do not match, the foreign key join candidate is not valid and will be discarded by the system. The surviving inter-database foreign key joins are added to the schema graphs of the local databases, and an approximate global schema graph is built.

Online query processing follows schema graph-based single database keyword search but extends it in the following ways to improve the scalability. Initially, every table R in the integrated database is searched for tuples that contain some of the keywords in Q. The retrieved tuple sets and the integrated schema graph are used to create a tuple set graph, which describes every way tuples may be joined within and across databases. Based on the tuple set graph, candidate networks, i.e., trees satisfying a given size threshold, are generated, and materialized subsequently using distributed SQL queries. In order to improve the scalability, the tuple set graph is condensed in that way, that different joins between the same two nodes are collapsed into one join. Now, the system generates condensed candidate networks. The candidate networks are evaluated using iterative refinement, i.e., only for the top-k necessary parts of the network are materialized. Condensed candidate networks and iterative refinement allow better scalability for search across different sources.

Answering keyword queries over virtual views

The KITE system [SLDG07] automatically extracts connections between different databases. Such connections can be constructed manually using views. Shao et al. [SGB+07] presented a system that implements efficient keyword search over Virtual XML Views. The problem of keyword queries over virtual views imposes two main challenges. A view combines different sources with join connections. Therefore, the query keywords may be distributed over different sources and only the combined sources contain the result. The views have to be materialized to obtain the connections which is inefficient. The first problem is the selective extraction of elements that provide information about the connection without materializing the complete view. The second task is the efficient generation of scoring statistics from the base data, so that the ranking is the same as in the materialized view. Shao et al. proposed the following system architecture (see Figure 4.11) and a corresponding three-phase algorithm.

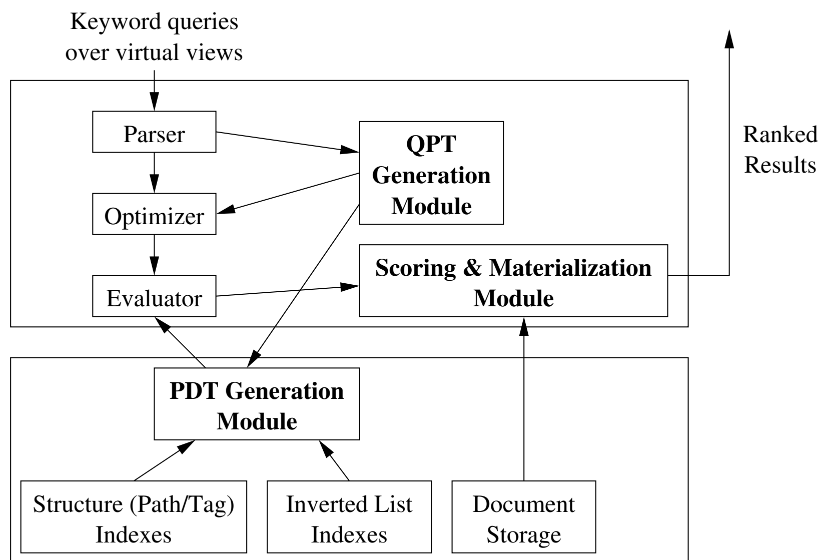


Figure 4.11.: Querying over virtual XML views [SGB+07]

Initially, a keyword query over virtual views is parsed. Using the keyword query and the view definitions, one query pattern tree (QPT) for each source is created by the QPT generation module. A QPT comprises all structural and content predicates that are necessary for query evaluation, i.e., join connections, and result materialization. A QPT is a tree in which the nodes contain search predicates, e.g., tag names, view predicates, or keyword predicates. Edges represent either parent-child or ancestor-descendant relationships. Furthermore, nodes and edges are annotated describing whether they are necessary for query evaluation and/or result materialization. The QPT is sent to the PDT (pruned document tree) generation module. Furthermore, the original query is modified and sent to the standard query evaluator.

The PDT generation module creates pruned document trees that contain only elements that satisfy the QPT, i.e., that are necessary for query evaluation. The generation module uses only indexes on the base data for PDT construction. Furthermore,

the PDT contains all statistic information about the terms, which are required for score computation.

The standard query evaluator uses the constructed PDTs as well as the modified queries of QPT generation module to execute the view queries and to combine the PDTs. Finally, the Scoring and Materialization module retrieves the actual data from the base data and returns the ranked results.

In summary, the approach of Shao et al. [SGB+07] allows the efficient keyword search over virtual XML joins. In that way, it supports distributed, cooperative, virtual integrated data sources. Limited heterogeneity can be resolved using view definitions.

Natural Language Search over Mediator Systems

Analogously to natural language queries (NLQ) over single structured databases, NLQ can also be applied to mediator systems integrating heterogeneous structured databases. Liu et al. [LLY+05] proposed an NLQ system on top of a mediator system [SBJ+01]. In the proposed system, databases are represented as 3-tuple $DG = (V, E, P)$. The set of nodes V contains relation nodes, attribute nodes as well as value nodes that represent a set of corresponding attribute values. The set E is a set of edges between nodes comprising containment edges as well as foreign key relationship edges. Every edge has an assigned weight describing the semantic distance between two nodes. Finally, P is a set of labeled paths. A path is a set of nodes that are connected by edges. The label describes the semantic relationship between start and end node, e.g., Be, Of, or $Related$. Given a set of databases represented as described above and an NLQ, the result is computed using the following steps:

1. A NLQ query q is analyzed and transformed into a directed, labeled query graph $QG = (V, S, E)$ where V is a set of nodes representing the terms of q, the set $S \subset V$ represents target nodes, i.e., the desired output of the query, and E is a set of labeled edges between the nodes. The edge label indicates the semantic relationship between two query nodes. In detail, the labels are be, $have$, of, $possessive$, and $attached$.

2. The second step is the database ranking, i.e., the selection of sources that fit best to the query. For this purpose, the query graph QG is mapped to a database graph DG. Firstly, every node of QG is mapped to a node of DG using a lexicon. The lexicon stores triples, which relate possible terms to database schema nodes and the corresponding semantic distance. An ontology of words is used to extend the query and to improve the mapping. Secondly, every edge in QG is mapped to a database path in DG using a mapping table. Finally, the mapped sub-graph of DG is an answer graph. The best result of a mapping is the answer graph with the least sum of edge weights. In summary, the databases are ranked by increasing semantic distance.

3. In the third step, the answer graphs are translated into queries that are sent to the sources. The results are retrieved, ranked and presented to the user.

In summary, the approach of Liu et al. implements natural language queries on heterogeneous structured databases by reusing the functionality of a mediator system. The system is capable to select best databases. Results do not span across different sources in contrast to the previous two discussed systems.

Integration by querying

Typical mediator systems distinguish between integration and query phase. During the integration phase, local schemas are integrated, i.e., correspondences are discovered, integration conflicts are resolved, and a global schema is created. That classical approach conflicts with fast-changing information needs and user-specific point of views to the data. Therefore, Domenig and Dittrich proposed the system SINGAPORE [DD01b] and query-based integration of heterogeneous data sources [DD00]. On the one hand, query-based integration presupposes a query-language that allows fuzzy as well as exact queries. On the other hand, a user and the system need as much as possible meta information about the sources to decide which data is expected and to translate global queries into local source queries. Domenig and Dittrich address these issues in the SINGAPORE system, which follows the three-layer architecture of mediator systems but provides a global query language and a union of local schemas instead of a global, integrated schema.

Data model. The local data is regarded as class extension and is considered as a set that can be queried. Furthermore, in several levels more general data sets are constructed which combine data sets from one or several sources. In fact, the data space forms a tree where parent nodes represent the union of the data sets that are assigned to the descendant nodes. There is one exception, nodes that describe the structure of the content in the data sources are considered as composition nodes, e.g., a relational table is composed of its attributes. In the data space, every internal node represents a set and can be queried. Figure 4.12 illustrates the data space that is structured into four levels. Level 1 contains the structure of the local sources. Level 2 comprises the source names. The type of the source (structured, semi-structured, unstructured) is described in level 3. Level 4 contains a classification hierarchy in which the sources are classified.

Query language. OQL is extended to express fuzzy queries over the data space by following concepts:

- the `contains` operator allowing keyword alike queries in the WHERE clause,
- regular path expressions as in semi-structured query languages [Abi97],
- operator `LIKE` allows the keyword search in the data space, i.e., in the metadata, so that fuzzy structural queries are possible.

In that way, the user is enabled to express queries without complete knowledge about the data space.

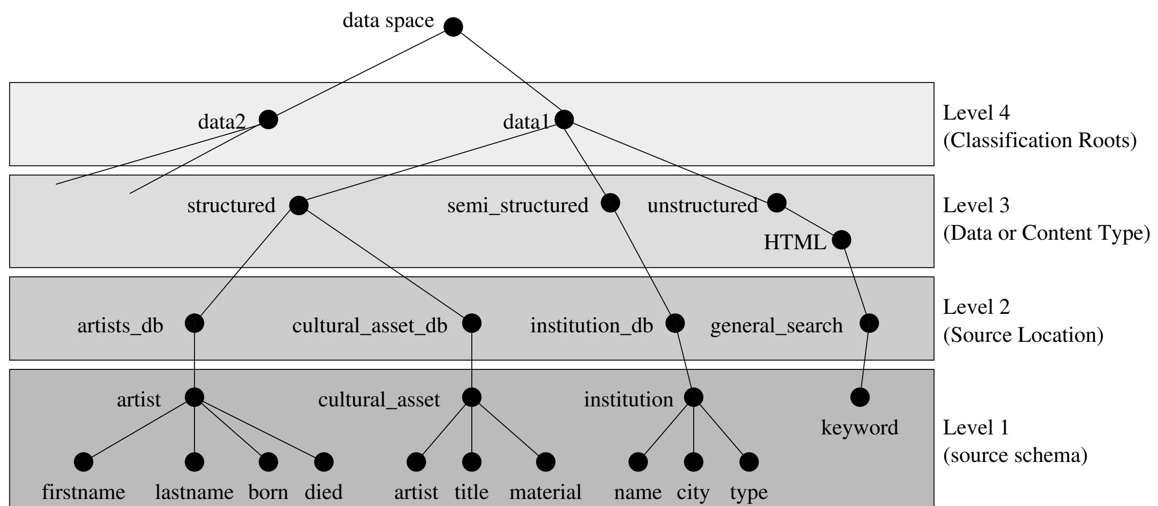


Figure 4.12.: SINGAPORE data space [DD01b]

Example 4.8 *Given the data space in Fig. 4.12, the user can formulate a query* `SELECT * FROM LIKE(painter.name) WHERE CONTAINS("Gogh")`*, in order to get all information about painter whose name contain the word "Gogh" in any sub-node.*

Metadata repository. The collected metadata is used to support the source selection by the user and to enable the system to rewrite the fuzzy queries. The metadata in the data space includes

- information about the structure, query capabilities, and result types of the sources,
- soft metadata, i.e., descriptive information about sources in text form as well as textual information about correspondences between sources and integration conflicts,
- ontological knowledge, i.e., a thesaurus that provides information about synonyms of search keywords.

The content of the metadata repository is indexed using an inverted index. The index consists of normalized keywords and posting lists that contain all paths that start with the keyword and all paths that end with the keyword and start with a data source name. The former case is used for keywords that occur in level three or higher of the data space, and the second if the keyword appears in level 1. Given this metadata repository structure, SINGAPORE processes global, fuzzy queries as follows.

Global query preprocessing. The system has to translate fuzzy global queries to a set of exact queries in the first step [DD01a]. First, the fuzzy paths in the `LIKE` operator are processed. Given an input path, for each component of the path, it is considered as keyword and all paths containing the keyword are obtained from the metadata repository. Subsequently, the obtained path sets are intersected to create

a set of exact candidate query paths. In the next step, the `contains` operator is rewritten taking into account the source query capabilities. Finally, the query parts are combined and tested for consistency. Consistent queries are translated into source queries and executed. In that way, SINGAPORE uses keyword search in the metadata repository to rewrite fuzzy path queries into exact queries.

Further approaches

We now review exemplary further studies that cover other problems querying heterogeneous sources for keywords. Marian et al. investigate top-k queries according to a monotonic ranking function over sources with different query capabilities [MBG04]. The authors assume two kinds of Web sources. The first kind returns a sorted list of results according to a query. The second kind uses a random access to retrieve the score of an object for an attribute. Now, the scores of different sources have to be combined to the final score. The authors adapt Fagin's TA algorithm [FLN01] and provide an alternative based on upper bound scores and fast reduction of scores to decide if a result is in the top-k result. Marian et al. study thereby vertically split relations, i.e., every source provides ranking one attribute of the object. In contrast, Hristidis and Papakonstantiou investigate the problem of horizontally split relations [HP04]. Different sources provide ranked results and we have to union the result for top-k processing. These works use also monotonic ranking function and follow the basic idea like keyword search on relation databases. Other works of joining results from different Deep Web sources and top-k query evaluation are [HML09] and [WAJ08, WA10].

4.5. Summary

There is a large body of research for keyword search on structured and semi-structured databases. This shows that the problem is recognized by the database community and it is important for making databases more usable. There are many different problems to be solved that are caused by the distribution of information across different data elements. We classified the systems according to the dimensions query language, result type, scoring and ranking methods, and processing method.

In the second part, we reviewed research of keyword search over distributed, heterogeneous data sources. Except few studies, most approaches deal with the selection of sources and merging of results, but not with the problem of combining data *across* different sources. The KITE system allows the search across different relational databases but assumes unrestricted query interfaces, i.e., all database operations are allowed. It does provide connections between sources but does not provide data and schema integration. We argue in this work that we require an integrated view to the data. In order to provide the integrated view, we use a concept-based mediator system. Furthermore, we require only limited querying capabilities of sources.

From the results of this chapter, we conclude following points for the keyword search approach:

- We require keyword-only and labeled keyword queries as query language, because they are simple to use and allow a restriction of results in the concept graph.

However, because of a complex schema, we must provide a kind of relaxation of the labels to allow effective search and to avoid too restrictive queries.

- A concept-based mediator approach distributes information across different concept instances that are connected by relationships. Hence, the result type of keyword search should be a tree of concept instances that answer together a keyword query.
- A ranking function should include a content score, a schema score, and should score the structure. For schema score, we must consider the local schema information, because users might only familiar with local schemata.
- As we require virtual integration, we do not assume a materialized data graph of global instances. Instead, we materialize the data during querying. Hence, we argue that the schema graph-based approach is the choice for combining a mediator system and keyword search. That means, we need keyword statistics on the global level that describes the keyword membership to concepts and properties, but not in global instances. This requires limited cooperation of the sources.

In the following chapter, we define keyword search over a concept-based integration model. Subsequently, we present the keyword processing approach following the schema graph-based approach.

5. Concept-based Keyword Queries

We showed in Chapter 2 that the integration of heterogeneous, semi-structured data sources is a necessity nowadays. There are many tasks to fulfill this integration task. We emphasized that concept-based mediator systems allow the integration of semi-structured, structured, and web-based data sources. We presented the concept-based mediator system YACOB in Chapter 3. Concept-based mediator systems are especially useful for few but complexly structured sources. The systems provide access to virtually integrated sources. However, concept-based mediator systems also provide complex, domain-specific concept schemata and powerful, complicated query languages. Simpler query interfaces like keyword-based queries are required.

Chapter 4 described keyword-based query systems over structured and semi-structured databases that also have the goal of easy access to structured and semi-structured databases. However, most approaches are defined for relational and XML data as well as for centralized databases. This chapter defines keyword queries over a concept-based global schema and virtually integrated sources. In that way, the chapter provides the basis for the subsequently presented keyword processing. Figure 5.1 shows the contribution of the present chapter in reference to the complete work.

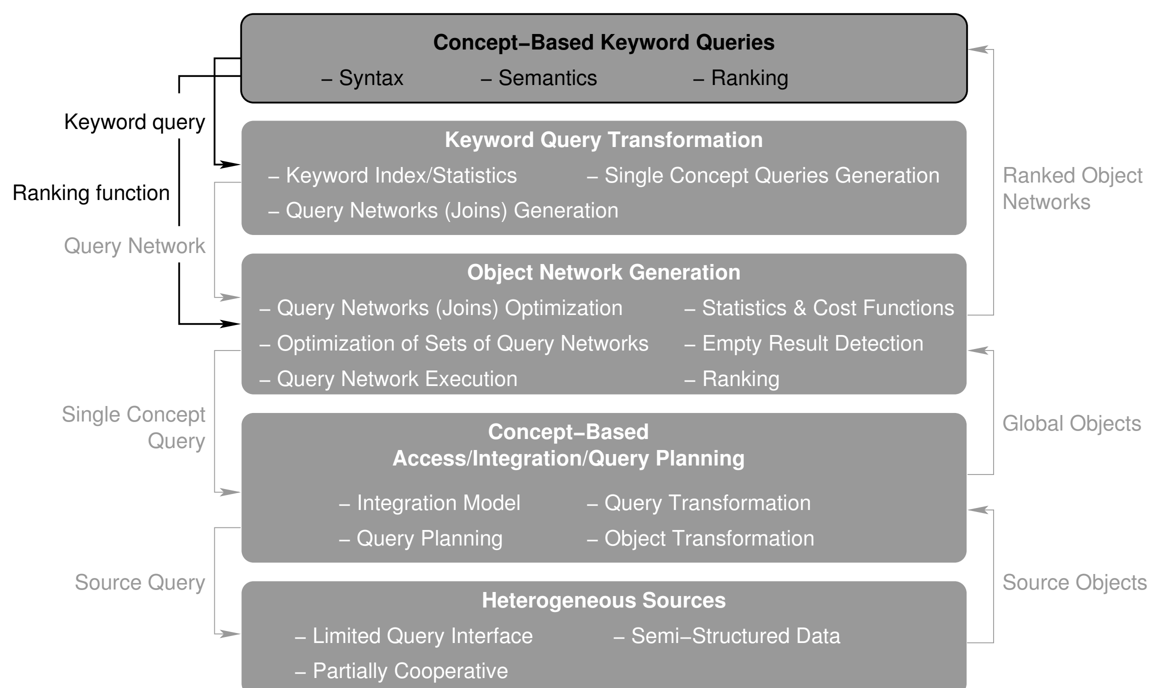


Figure 5.1.: Overview keyword semantics

The present chapter defines *concept-based keyword queries* and *object networks* as their results. Concept-based keyword queries are labeled keyword queries over a concept schema graph. Object networks are connected objects from different concept extensions that answer a concept-based keyword query, i.e., contain all or some query terms. The corresponding concept schema defines the connections between the objects with concept properties. We rely in the discussion on the YACOB system, presented in Chapter 3. However, the definitions can be adapted to other systems. We address the following points in this chapter:

Data model: The data model uses the YACOB model as the basis. We describe connected instances of concepts, denoted as object networks, as results of keyword queries.

Keyword query syntax and semantics: After specifying the syntax of plain and concept-based keyword queries, we describe the semantics of keyword queries as lists of object networks that satisfy the queries and are ordered by query score. Thereby, we discuss different semantics based on the structure of the result.

Ranked materialization queries: The mediator virtually integrates data and objects are integrated during querying. That means that we define materialization queries that create valid result object networks for given keyword queries. We also provide a scoring function that ranks materialization queries according to a keyword query and virtual database. Thereby, we exploit limited statistics about keywords and sources.

The chapter starts with a motivation of keyword queries and describes the challenges arising from the keyword search processing.

5.1. Motivation

This section gives four introducing examples showing the necessity of general keyword and concept-based keyword queries in the environment of a concept-based integration system. We consider the YACOB system and the following scenario. There is a set of databases, e.g., Web databases, relational databases, XML systems, etc., that provide information about cultural assets lost in the World War II because of looting or vicissitudes of war. Furthermore, there are sources that provide information about artists and institutions like archives and museums, respectively. The mediator system integrates these sources. The system provides a concept-based schema for integration as well as usage. Figure 5.2 shows an exemplary part of the schema consisting of three concept hierarchies: cultural assets, institutions, and persons. The concepts have different properties and are connected by concept properties. Furthermore, the figure illustrates parts of several concept extensions. Figure 5.3 shows the details of exemplary objects.

The following examples show typical queries that can be easily expressed as keyword queries, but are difficult to express in CQuery or other concept-based query languages. The queries also require deep knowledge about the structure of the data.

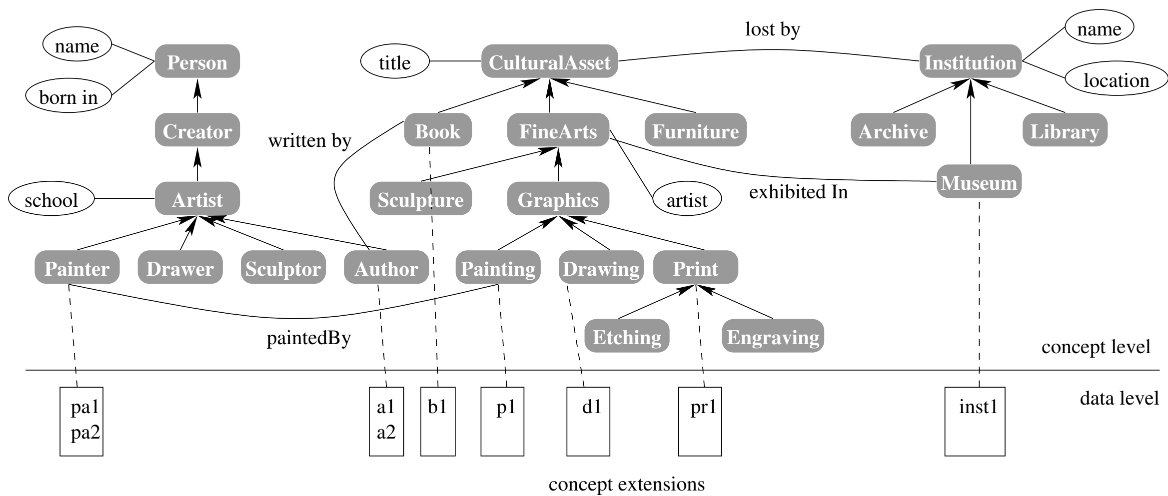


Figure 5.2.: A concept schema example consisting of three concept hierarchies. Concepts are illustrated as boxes and properties as ovals. Concept properties are named lines.

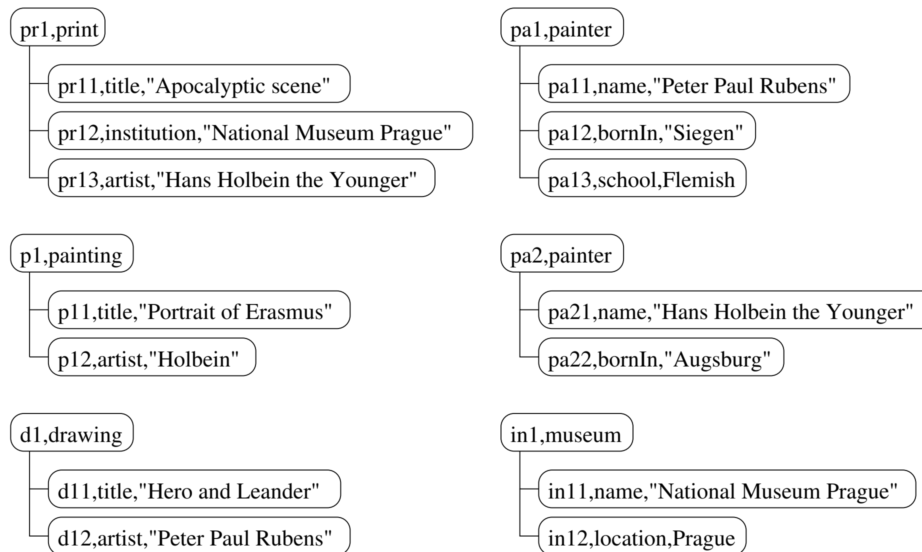


Figure 5.3.: Exemplary objects

Example 1. The user wants to retrieve all paintings made by Peter Paul Rubens. She knows the corresponding concept (paintings) but not the exact property, hence, the user can issue the CQuery statement illustrated in Figure 5.4. However, the system cannot optimize this kind of `WHERE` condition and searches all properties of concept painting for the keyword. If we add more keywords to the `WHERE` condition, the system will generate all combinations of properties and keywords. The same query could be expressed using a labeled keyword query [HN02, CMKS03]. In this case, we have two labels: the first corresponds to the concept and the second to the property. Assuming the concept is known to the user, we can formulate the keyword query "`painting::rubens`" with empty property label.

```
1  FOR $c IN concept[name="Painting"]
2  LET $e := extension($c),
3       $p := $c/properties()
4  WHERE $e/$p ~= "rubens"
5  RETURN $e
```

Figure 5.4.: Listing CQuery keyword example

Example 2. The user looks for objects that contain the keywords "artist rubens" and does not give any further structural information. The system has to decide where to search: in schema elements or data values or both. Possible results are paintings, where the keyword "rubens" occurs in the property "artist" or instances of the concept "artist" that contain the keyword "rubens" in the "name" property. We cannot efficiently express such queries in CQuery. Therefore, we have to find a way to translate the keyword query into CQuery or similar statements. These statements retrieve instances from one concept without combining different objects. This kind of keyword queries is supported by the Yacob system. It was introduced by Declercq [Dec04] and also described by Sattler et al. [SGS05].

Example 3. In the third example, the user knows approximately the concepts where required objects occur. Unfortunately, she cannot specify the connection between them. For example, consider the query "painter:name:holbein institution:location:prague". The user would like to get information about the painter Holbein and his works presented in or lost by institutions in Prague. In this case, two problems occur. Firstly, the system has to find instances of the concepts "painter" and "institution". Thereby, the system has to consider also sub-concepts and semantically related concepts. That means that we need query expansion or relaxation. For example, an artist Holbein may also be classified as an illustrator or sculptor. Secondly, the system has to find the connection between these instances. That task is denoted as proximity search [GSVGM98, HKPS06, LYJ08]. The instances are connected either directly or indirectly through instances of other concepts. For example, an artist instance "Hans Holbein the younger" is connected to the print instance "Apocalyptic scene", because it was made by him (see Figure 5.3). Furthermore, an institution "National Gallery" in Prague exhibits the print "Apocalyptic scene" that was stolen in WW II. Thus, the system combines the three instances to an answer with respect to the original query. We denote these answers as object networks.

Example 4. The last example illustrates the most general case. The user issues a plain keyword query and does not or cannot provide any further information. For example, the query "rubens cultural assets Flemish" might retrieve a set of object networks that contain information about an artist "Rubens", cultural assets lost in a "Flemish institution", but also information about art made by a Flemish artist named Rubens.

The keyword query processing has to combine all basic approaches above: location of the keywords, query expansion as well as automatic connection between objects

using joins. Similar approaches are keyword search in relational and semi-structured databases (see Sections 4.3.1 and 4.3.2), but these systems mainly support central databases and provide only limited schema search support. We reviewed in Section 4.4.3 keyword search over heterogeneous data sources. However, these approaches do not provide concept-based integration and schema element searches.

Based on the discussion of the examples, we will propose a new approach that combines concept-based integration and keyword search as techniques to enable users to search heterogeneous, semi-structured data sources. First, we define the data model for keyword search as well as the syntax of the queries and the query semantics on the basis of object networks. Second, a proposed scoring function ranks result object networks according to their relevance. Third, we define query expansion mechanisms that exploit the semantic relationships of the concept-based integration model.

5.2. Data Model

For keyword search, we generalize the YACOB integration schema and represent all textual information in sources and schema as *bags of terms*. While we use the YACOB system as foundation, other concept-based mediator systems with similar properties can be used, too. We model the global database as a *Virtual Document*. A virtual document is a directed graph. Every node represents a concept and its related information. The nodes are connected by two kinds of edges: the first set of edges models the **is_a** relationship between concepts; the second set of edges represents the set of concept properties. We illustrate three connected nodes of a virtual document in Figure 5.5. The terms of the nodes and the properties are extracted from the global labels as well as from the source descriptions. Furthermore, the figure illustrates partial source content terms for three exemplary properties.

In detail, every node n in the virtual document has a label *n.label*, a bag of associated terms *n.terms*, and a set of properties *n.properties*. The label *n.label* identifies the node. If we consider YACOB as the underlying model, the label will be the URI of the concept. For example, the label *painter* identifies a node that represents the concept "painter" (see Figure 5.5). The bag of terms *n.terms* contains all terms that describe the node, i.e., the concept. The terms are extracted from the label name, the URI, and the source mappings, for example. In the given example, the terms stem from the global label ("painter") and mappings ("artist", "painter", etc.). The set of properties *n.properties* contains information about literal and categorical properties.

A property p consists of a label *p.label*, a bag of terms *p.terms*, and a set of source terms *p.sources*. The label *p.label* is the URI of a property, and *p.terms* contains all terms of the concept level assigned to the property (label, URI, source descriptions). For example, the concept "painter" has the property "name". Thus, the properties set of the node contains a property representation with $label = "name"$ with the bag of terms $\{name, name\}$ because the global property is mapped to a local property that is also denoted as "name".

The set of source terms *p.sources* contains pairs $(s, T(s, p))$ where s is a source and $T(s, p)$ a bag of terms. $T(s, p)$ consists of terms found in the property value of p in objects of the extension of the corresponding concept of n. The terms in

$T(s, p)$ represent the extension of the concepts, i.e., the set of all instances. In case of a categorical property, the corresponding source terms are categories, expressed by global categories. In the example, the property "name" of concept "painter" represents two sources, "webart" and "lostart". Figure 5.5 illustrates the respective bag of terms $T(webart, name)$ and $T(lostart, name)$.

The edges of the virtual document indicate subclassOf relationships $E_{\mathbf{is_a}}$ and concept properties E_{lnk}. A concept property edge e has a label $e.label$ that is the URI of the underlying concept property. The source node $e.src$ represents the domain of the property, and the target node $e.tgt$ the range. Furthermore, the edge has an assigned bag of terms $e.terms$ containing all schema terms assigned to e. At last, the edge has cardinalities $e.srcCard$ and $e.tgtCard$. The cardinalities indicate the type of the edge: one to one, one to many, many to one, or many to many. A subclassOf edge does not have any attributes.

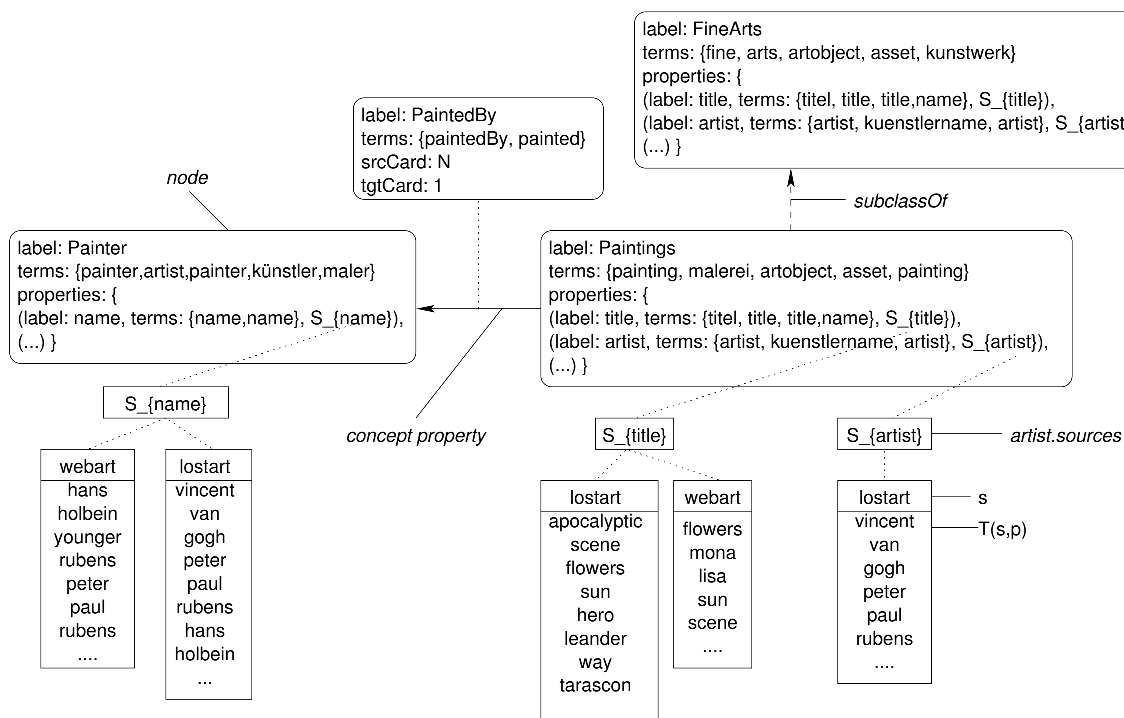


Figure 5.5.: Virtual document

The following definition summarizes the virtual document. Let $\mathcal{T}$ be the set of all terms and $MS(\mathcal{T})$ the set of all bags of terms.

Definition 5.1 Virtual Document
A virtual document is a directed, labeled graph $D = (N_D, E_D)$ with

- *N_D is a set of nodes with $\forall n \in N_D$:*
  - *$n.label \in URI$ the identifying label of the node,*
  - *$n.terms \in MS(\mathcal{T})$ a bag of terms,*
  - *$n.properties$ a set of property information.*

- *A property $p \in n.properties$ consists of*
  - $p.label \in URI$ *the identifying label of the property,*
  - $p.terms \in MS(\mathcal{T})$ *a bag of terms,*
  - $p.source$ *a set of source content terms of the form* $(s, T(s,p))$ *with* $s \in Name$ *the name of the source and* $T(s,p) \in MS(\mathcal{T})$.
- $E_D = E_{\boldsymbol{is_a}} \cup E_{lnk}$ *is the set of edges with* $\forall e \in E_D$ $e.src \in N_D$ *the source node and* $e.tgt \in N_D$ *the target node. For each edge* $e \in E_{lnk}$ *the edge has following properties*
  - $e.srcCard \in \{1, N\}$ *and* $e.tgtCard \in \{1, N\}$ *the cardinalities of the concepts,*
  - $e.label \in URI$ *the URI of the concept property,*
  - $e.terms \in MS(\mathcal{T})$ *a bag of terms.*

□

For each virtual document, a concept graph exists. The *concept graph* is the graph of all node labels, connected by labeled edges and subConceptOf relationships. The concept graph acts as schema of the virtual document describing the concepts only. We define the concept graph as follows.

Definition 5.2 Concept Graph
A concept graph *of a virtual document* $D = (N_D, E_D)$ *with* $E_D = E_{\boldsymbol{is_a}} \cup E_{lnk}$ *is a directed graph* $CG(D) = (C_{CG}, E_{CG})$ *with:*

$$\begin{aligned} C_{CG} =& \{n.uri | n \in N_D\} \text{ and} \\ E_{CG} =& \{(n_1.uri, n_2.uri, e.label) | e = (n_1, n_2) \in E_{lnk}, \} \\ & \cup \{(n_1.uri, n_2.uri, \boldsymbol{is_a}) | (n_1, n_2) \in E_{\boldsymbol{is_a}}\}. \end{aligned}$$

□

The concept graph represents the concept schema of the virtual documents without literal and categorical properties.

Handling categorical values

Categorical values are homogeneously modeled as category hierarchies on the global level in Yacob. Every category is represented by different values in the sources and a global name. We now extend the virtual document by separate category hierarchies. The categories create a directed, acyclic graph $V = (K_V, E_V)$. This graph is a set of not connected trees. The nodes represent categories. The directed edges model the `subCategoryOf` relationship between categories. Every node $n \in K_V$ has an identifying label *n.label* $\in$ *URI* and a bag of terms *n.terms* $\in MS(\mathcal{T})$. The bag of terms consists of the global name of a category as well as all local category notations.

We extend the Definition 5.1 here and add the graph V to the virtual document. That means that a virtual document is defined as

$$D = (N_D, E_D, V).$$

Concept and object network

We denote a view on a concept graph as a *concept network*. A node in the concept network consists of an alias and the label of the concept node. The edges are labeled with the label of the edge between the concept nodes in the concept graph. Furthermore, the concept network does not contain subConceptOf relationships.

Definition 5.3 Concept Network
A concept network *is a directed, acyclic, labeled graph* $CN = (A_{CN}, E_{CN})$ *with nodes* $A_{CN} \subset Name \times URI$ *and* $E_{CN} \subset A \times A$*. A node* $(alias, uri) \in A_{CN}$ *consists of a unique alias and concept node label uri. An edge* $e \in E_{CN}$ *has a label* $e.label \in URI$*. A concept network* $CN = (A_{CN}, E_{CN})$ *is a view to a concept graph* $CG = (C_{CG}, E_{CG})$ *if*

- $\forall (alias, uri) \in A_{CN} : uri \in C_{CG}$*, and*
- $\forall e = (a_1, a_2) \in E_{CN} \exists e' = (uri_1, uri_2) \in E(CG) : uri_1 = a_1.uri \wedge uri_2 = a_2.uri \wedge e.label = e'.label$.

□

At last, we define an object network ON. An object network is a set of objects that are connected via concept properties, i.e., they belong to the join of the corresponding concept extensions. While objects are instances of a concept extension, object networks are instances of a concept network. Thus, an object network is a graph where nodes are objects and edges conform to concept properties that connect the objects. Every concept property is mapped to a join. Therefore, two connected objects form a tuple that belongs to the join. Assume a mapping function $map : \mathcal{O} \rightarrow Name \times URI$ with $map(o) = (a, uri)$ if $o \in \mathbf{ext}(uri)$.

Definition 5.4 Object network
An object network *is a directed, acyclic, labeled graph* $ON = (O_{ON}, E_{ON})$ *with* $O_{ON} \subseteq \mathcal{O}$ *and* $E_{ON} \subset O_{ON} \times O_{ON}$*. Every edge* $e \in E_{ON}$ *has a label* $e.label \in URI$*. The object network* ON *conforms to a concept network* $CN = (A_{CN}, E_{CN})$ *if*

- $\forall o \in O_{ON} \exists a \in A_{CN} : map(o) = a$ *and for all pairs of distinct objects* o, o' *with* $o \in O_{ON}$ *and* $o' \in O_{ON}$ *and* $o \neq o'$*, it holds* $map(o) \neq map(o')$*, and*
- $\forall e = (o, o') \in E_{ON} : (o, o') \Leftrightarrow e' = (map(o), map(o')) \in E_{CN}$ *with* (o, o') *is an instance of the join* $\boldsymbol{ext}(a.uri) \bowtie_{JM(e')} \boldsymbol{ext}(a.uri)$.

The size of the object network is defined as the number of objects $\text{size}(ON) = |N_{ON}|$.
□

Example 5.1 *Consider the figures 5.2 and 5.3. A possible object network is*

$$d1[(paintedBy, pa1), (exhibitedIn, in1)]$$

saying the work "Hero Leander" (object d1) was painted by "Rubens" (pa1) and is exhibited in the museum "National Museum Prague" (in1). The corresponding concept network would be

$$CN = (a1, drawing)[(paintedBy, (a2, painter)), (exhibitedIn, (a3, museum))].$$

A second object network

$$pr1[(paintedBy, pa2), (exhibitedIn, in1)]$$

describes that the print "Apocalyptic scene" (pr1) was made by "Holbein" (pa2) and is exhibited in the war by the museum "National Museum Prague"(in1).

In keyword search, we find object networks that satisfy a concept-based keyword query. In the following section, we define the keyword queries and their semantics based on materialization queries.

5.3. Query Model

Based on the data model, we now define the query model of the keyword search system. First, we introduce the keyword query syntax. Second, we define minimal object networks as results of keyword queries. Third, we define materialization queries that are used to create valid results. Fourth, we give three semantics of keyword queries based on materialization queries and a ranking function.

5.3.1. Keyword Queries

In this work, we define a keyword query as a set of *query terms* $Q = \{t_1, t_2, \ldots, t_{|Q|}\}$. We distinguish two classes of query terms and keyword queries: plain keyword queries and concept-based keyword queries.

A *plain keyword query* only contains plain keywords as query terms. That means that every term $t_i \in Q$ is an element of the set of terms $\mathcal{T}$. Plain keyword queries do not restrict the position where the keywords can occur. In particular, all query terms may match concept level and data level elements.

We denote the second class of keyword queries as *concept-based keyword queries*. A concept-based keyword query belongs to the class of labeled keyword query [CMKS03, YJ07]. The search terms consist of a label and a value keyword. The label determines in which types of objects the system has to search for value keywords. Concept-based query terms consist of two label keywords and one value keyword. The first label keyword refers to the concept, and the second label describes the property. Both label keywords are evaluated over the concept level, i.e., the concept schema and the source descriptions. The value keyword matches data level object values. In summary, we

define a concept-based query term t as a triple of keywords $t = (kw^c : kw^p : kw^v)$. Note, that every part of a concept-based query term may also be empty, i.e., it can have the value *null*.

Definition 5.5 Keyword Query
A set of query terms $Q = \{t_1, t_2, \ldots, t_{|Q|}\}$ *is either a*

1. plain keyword query *if* $t_i \in \mathcal{T}$, *for* $1 \leq i \leq |Q|$, *or*

2. concept-based keyword query *if* $t_i = (kw_i^c : kw_i^p : kw_i^v)$ *with* $kw_i^c, kw_i^p, kw_i^v \in \mathcal{T}^*$ *for* $1 \leq i \leq |Q|$ *with* $\mathcal{T}^* = \mathcal{T} \cup \{null\}$. *Given the set of role* $ROLE = \{concept, property, value\}$, *we say that for all* $t_i \in Q$, *keyword* kw_i^c *has the role "concept", keyword* kw_i^p *the role "property", and keyword* kw_i^v *the role "value", respectively.*

□

Example 5.2 *The following keyword queries are typical examples. The query*

$$Q_1 = \{(painting : title : sunflower), (painting : artist : Gogh)\}$$

searches for paintings where the title contains "sunflower" and the artist name contains the keyword "Gogh". The second query

$$Q_2 = \{(painting : null : sunflower), (artist : null : Netherlands)\}$$

does not specify property keywords in the concept-based query terms. It searches for paintings that contain the value keyword "sunflower" and are connected to artists from the "Netherlands". The last exemplary query

$$Q_3 = \{painting, sunflower, artist, Gogh\}$$

is a plain keyword query containing the same keywords but does not specify the role of the keywords.

5.3.2. Minimal Object Networks

In Section 5.1, we showed that keywords of a query are spread over different connected objects. This makes the result of keyword queries to be a list of object networks. At this point, we assume there is an operator *contains* that decides if an object o or an edge e contains a query term t. Thereby, t is either a plain query term or a concept-based query term. We define the *contains* operator and materialization queries in the following Section 5.3.3.

With the help of this operator, we can define a minimal object network according to a keyword query. As shown in Hristidis et al. [HGP03], result networks have to satisfy the minimality condition to avoid spurious results.

Definition 5.6 Minimal Object Network
A minimal object network $ON = (O_{ON}, E_{ON})$ *is a result according to a plain or concept-based keyword query* $Q = \{kw_1, kw_2, \ldots, k_{|Q|}\}$ *and a number* $size_{max}$ *if*

- *all (AND) or some (OR) keywords are contained in the object network,*
- *all leaf nodes (objects) of the network contain at least one search keyword or are connected to the network by an edge that contains at least one keyword,*
- *the network size* size(ON) *is not bigger than* $size_{max}$*, and*
- *the network is minimal, i.e.,*

  AND case: *no object or edge can be removed without breaking the previous rules or disconnect the network.*

  OR case: *no object or edge can be removed without removing a keyword match, breaking the previous rules, or disconnect the network.*

□

The definition states the following points. Firstly, the object network has to contain all (or some) query terms using the operator *contains*. Secondly, the leaves of the network have to contain a keyword. Without this requirement, the networks can be extended arbitrarily without matching a keyword query better. This leads to spurious results. We extend previous definitions [HGP03] in that way, that we allow keyword containment in edges. If a leaf object is connected to the rest of the network with an edge that contains a query term, the leaf satisfies the minimal requirement. Thirdly, we require a maximum value of the object network size, denoted as $size_{max}$, to restrict the result set. We assume all networks with a bigger size are not relevant. In summary, all object networks satisfying the defined characteristics are valid results according to a concept-based keyword query, a maximum size, and the containment operator.

For example, Figure 5.6 illustrates two minimal object networks. The first is an answer to the query $\{prague,\ augsburg\}$. The second network is an answer to the query $\{Flemish,\ Leander,\ lost\}$. It is minimal, because the object $in1$ is connected with the edge $lostBy$ that contains the keyword $lost$.

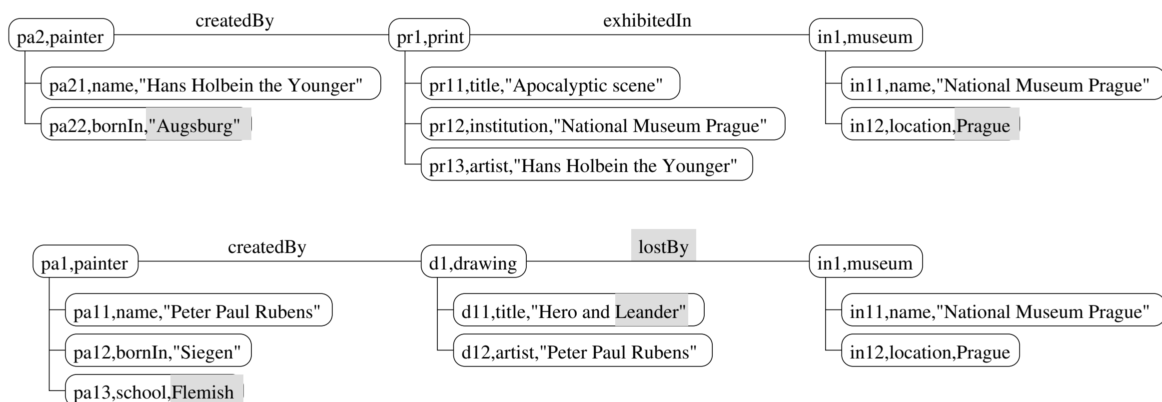


Figure 5.6.: Exemplary minimal object networks

5.3.3. Materialization Queries

Objects and object networks are not materialized but have to be created in order to be an answer of a keyword query. Therefore, we use the virtual document as the basis to create *materialization queries* according to a keyword query. A materialization query is used to obtain and combine global objects. A materialization query is a structured, concept-based query and an interpretation of the keyword query over the virtual document. It guarantees that its results are valid object networks according to the keyword query. However, a materialization query cannot guarantee that its result set is not empty. Thus, a materialization query corresponds to keyword query interpretations as described in the systems SQAK [TL08] or IQ^p [DZN10].

Given is a virtual document $D = (N_D, E_D, (K_V, E_V))$ that provides the information of positions of keywords but not object identifiers. Based on the position, a keyword $kw \in \mathcal{T}$ is interpreted in the following way.

Definition 5.7 Keyword Interpretation
The keyword interpretation $(kw, uri, pred)$ *of* $kw \in \mathcal{T}$ *with respect to* D *has a predicate* $pred$ *of the form*

Case 1: $pred := true$ *if*

Case 1a: $\exists n \in N_D : n.label = uri \land kw \in n.terms$

Case 1b: $\exists e \in E_D : e.label = uri \land kw \in e.terms$

Case 2: $pred := \mathtt{exists}(prop)$ *if* $n \in N_D\ \exists p \in n.properties : n.label = uri \land p.label = prop \land kw \in p.terms$,

Case 3: $pred := prop \sim= kw$ *if* $n \in N_D\ \exists p \in n.properties\ \exists (s, T(s,p)) \in p.sources : n.label = uri \land p.label = prop \land kw \in T(s,p)$,

Case 4: $pred := prop = v.uri$ *if* $n \in N_D\ \exists p \in n.properties\ \exists (s, T(s,p)) \in p.sources : n.label = uri \land p.label = prop \land v.label \in T(s,p)$ *with* $v \in K_V$ *and* $kw \in v.terms$.

□

The keyword interpretations are classified into two groups: schema level and data level interpretation. The schema level interpretations refer to schema level elements and do not filter the extensions (case 1) or test only for existence (case 2), respectively. The second group will select objects from an extension (cases 3 and 4). Case 1 assumes that a keyword is found in a concept (1a) or concept edge (1b). Thus, the interpretation does not induce a filter predicate. For example, the keyword "painter" is found in the terms of the node with the label "Painter" (see Figure 5.5). The corresponding interpretation is $(painter, Painter, true)$. Thus, we select a concept or concept edge with the keyword. In case 2, the keyword is found in a data property. It is also in the schema role, but also states, that retrieved objects must include a value for the property. For example, the keyword "title" is found as property term for data property "title" of node "paintings". The interpretation is $(title, paintings, \mathrm{exists}(title))$. Case 3 expresses that the keyword kw occurs in a data value in a source. For example, assume the keyword "Vincent". This term occurs in the lostart database for

the concept "painter" and the property "name". Therefore, we interpret the keyword as $(vincent, painter, name \sim= Vincent)$. The interpretation states that objects of the concept "painter" contain the keyword “vincent”. Case 4 handles categorical values. We assume a category $v \in K_V$. The keyword contains the term kw, i.e., $kw \in v.terms$. If the label $v.label$ occurs in any source of concept/property pair, we can interpret the keyword kw as category selection. For example, the keyword "vegetable" occurs in the category "Fruits/Vegetables". If now the category "Fruits/Vegetables" occurs in a source for “paintings/motif” then the keyword will be interpreted as $(vegetable, paintings, motif = "Fruits/Vegetables")$.

We denote the set of all interpretations of keyword kw with respect to D as interpretation$^*(kw^c, D, role)$. The parameter $role$ has the values c, p, v that restrict the valid interpretation of the keyword. A concept keyword must occur in a concept node term set (c, case 1a), a property keyword must occur in a data property or concept property term set (p, case 1b) or case 2). Finally, a value keyword must occur in a data value (v, cases 3 or 4). The parameter $role$ can be empty for considering all interpretations. After the interpretation of one keyword based on the virtual document information, we deal now with the interpretation of plain and concept-based query terms. A plain query term is one keyword, i.e., $t = kw$. The interpretation of a plain query term t is than

$$\text{interpretation}(t, D) = \left\{\{i\} | i \in \text{interpretation}^*(t, D)\right\}.$$

In general, we create singleton sets from the interpretations. That is necessary to support concept-based query terms in the same way. We define the interpretation of a concept-based query term $t = (kw^c : kw^p : kw^v)$ as the set

$$\begin{aligned}\text{interpretation}(t, D) = \{\{i_c, i_p, i_v\} | & i_c \in \text{interpretation}^*(kw^c, D, c),\\ & i_p \in \text{interpretation}^*(kw^p, D, p),\\ & i_v \in \text{interpretation}^*(kw^v, D, v) \wedge\\ & i_c.uri = i_p.uri \wedge i_p.uri = i_v.uri \wedge i_c.uri = i_v.uri \wedge\\ & i_p.prop = i_v.prop\}.\end{aligned}$$

In every interpretation set for t, all interpretation of the concept keywords must refer to the same concept. The property and value keywords have to refer to the same property. That leads to following points. First, if a component keyword has the null value, we will ignore the correspondent terms in the computation. Second, an edge can only be referred by a concept-based query term, if the value and concept keywords are null.

For example, consider the concept-based query term $t = (artist : name : gogh)$ and the exemplary virtual document in Figure 5.5 on page 90. One interpretation of t is $\{(artist, artist, true), (name, artist, exists(name)), (gogh, artist, name \sim= gogh)\}$. Taken together, the term t is interpreted as all objects of concept artist that contain the keyword “gogh” in the property “name”.

Materialization query

Now we can define a materialization query as an interpretation of a keyword $Q = \{t_1, t_2, \ldots, t_n\}$. For a given virtual document D, we get a set of interpretation sets I_{t_i} for every query term t_i. The structure of a materialization query is described by a concept network. The complete materialization query consists of a concept network and a valid combination $\mathbf{I}$ of term interpretations from I_{t_i}.

Definition 5.8 Valid Materialization Query
Given are a keyword query $Q = \{t_1, t_2, \ldots, t_n\}$ and its interpretations $I_{t_1}, I_{t_2}, \ldots, I_{t_n}$ with respect to a virtual document D. A ***materialization query*** *for Q is a pair $mq = (CN, \mathbf{I})$ with $CN = (A_{CN}, E_{CN})$ a concept network and $\mathbf{I}$ a set of interpretations. For every node $n = (label, uri) \in A_{CN}$ or edge $e \in E_{CN}$, we define the assigned interpretations as $\text{assign}(n) = \{i | i \in \mathbf{I} \wedge \exists (kw, n.uri, pred) \in i\}$ and $\text{assign}(e) = \{i | i \in \mathbf{I} \wedge \exists (kw, e.label, pred) \in i\}$. Assuming AND semantics, the materialization query mq is valid with respect to Q if:*

1. *$|\mathbf{I}| = |Q|$ and $\forall t \in Q \exists i \in \mathbf{I} : i \in I_t$,*
2. *for each leaf node n of CN it holds $\text{assign}(n) \neq \emptyset$ or it exists an edge $e = (n, n') \in E_{CN}$ with $\text{assign}(e) \neq \emptyset$,*
3. *it exists at least on leaf node $(label, uri)$ with a keyword interpretation of the form $(kw, uri, prop \sim= kw)$ or $(kw, uri, prop = v)$,*
4. *$|A_{CN}| \leq Size_{max}$, and*
5. *no interpretation, alias, or edge can be removed without breaking the previous properties.*

□

The first point of the definition expresses that the set of term interpretations $\mathbf{I}$ contains exactly one interpretation for each query term. Given the exemplary query $Q = \{hans, flowers\}$ the set $\mathbf{I}$ must have one interpretation from each I_{hans} and $I_{flowers}$. Let $\mathbf{I}$ be the set $\{\{(hans, artist, name \sim= hans)\}, \{(flowers, paintings, title \sim= flowers)\}\}$, for example. The second point requires that the leaf nodes must be specified, i.e., a keyword interpretation has to be attached to them. The materialization query selects objects and combines them to object networks. Object networks must have keywords in every leaf. Thus, the keywords must be in every leaf of the materialization query, too. This ensures selection of valid objects and the construction of valid object networks. The third point requires that at least one data source selection must be in the materialization query. Since we cannot get all values from a source or concept extension, we must provide at least one data selection condition. The fourth point limits the size of materialization queries. The fifth point ensures the minimality.

Example 5.3 *Assume the keyword query*

$$painter : name : rubens, museum :: prague.$$

A valid materialization query $mq = (CN, \mathbf{I})$ *is*

$$(a1, Paintings)[(paintedBy, (a2, Painter)), (exhibitedIn, (a3, Museum))]$$

with interpretation assignments

$$\begin{aligned}\text{assign}(a2) =& \{\{(artist, Painter, true), (name, Painter, exists(name)),\\ & (rubens, Painter, name \sim= "rubens")\}\\ \text{assign}(a3) =& \{\{(museum, Museum, true), (prague, Museum, city \sim= "prague")\}\}.\end{aligned}$$

Translation of materialization queries

We show the translation of a materialization query into a CQuery statement of the YACOB system for illustration of materialization queries and as an example. Every concept property and, therefore, every edge in a materialization query is mapped to a join condition. We denote the join condition of an edge e as $cond(e)$. Algorithm 5.1 sketches the transformation. Assume as exemplary materialization query mq above. In step one, we create all node-related clauses in the CQuery statement. For each alias $a \in A_{MQ}$, we create the concept selection (line 4), e.g., \$a2 `IN` concept[uri = "Painter"], and the extension function (line 5), e.g., \$a2_ext := extension(\$a2). Additionally, we create the selection clause based on the assignments (lines 7-17). If the assignment is of the form $(term, uri, true)$, we will omit it. For example, the assignments for $a2$ are translated to `$a2_ext/Name is not null and $a2_ext/Name ~= "rubens"`. In the following, the algorithm adds the join conditions (lines 20-22). For every edge, we obtain the condition $cond(e)$ and replace the concepts with the corresponding variable names, and add the condition to the `WHERE` clause. For instance, the edge paintedBy is represented by the condition `$a1_ext/artist = $a2_ext/name`. In line 23 of the algorithm, we detect unnecessary predicates in the `WHERE` clause and remove them. For example, the algorithm removes the predicate `$a2_ext/Name is not null` because it is implied by `$a2_ext/Name ~= "rubens"`. Furthermore, the algorithm ensures that all objects of the result are disjoint (line 24). Finally, we add all extension variables to the `RETURN` clause and combine all clauses to the final statement. Figure 5.7 shows the final CQuery statement of the example.

Keyword query semantics

Keyword query semantics describes the result of a keyword query Q for a virtual document D. We use a scoring function score(mq, Q) that assigns a score value to a materialization query mq according to the keyword query Q. The function is used to rank the results, i.e., materialization queries as well as their result object networks are ranked by the score of the materialization query. The details of one possible function are described in the following Section 5.3.4. In the following, we will discuss three semantics: All-$size_{max}$ semantics, Top-k semantics, and Top-k concept networks.

Algorithm 5.1 Translation: Materialization query to CQuery

```
    Input: MQ = (A_MQ, E_MQ, I, assign())
    Output: CQuery statement cquery
 1: function TranslateMQToCQuery(MQ)
 2:     FOR = "",LET = "",WHERE = "",RETURN = ""
 3:     for all a = (alias, uri) ∈ A_MQ do
 4:         append to FOR statement "$alias IN concept[uri = "uri"]"
 5:         set extension variable var = alias + "_ext"
 6:         append to LET statement "$var := extension( $alias)"
 7:         for all (term, uri, pred) in assign(a) do
 8:             if pred == exists(p) then
 9:                 add to WHERE "$var/p is not null"
10:             else if pred == p ~= t then
11:                 add to WHERE "$var/p ~= "t""
12:             else if pred == p = v then
13:                 /* assume a category index idx that is incremented */
14:                 add to LET "$k_idx := $var/p[uri = "v"]"
15:                 add to WHERE “$var/p = $k"
16:             end if
17:         end for
18:     end for
19:     /* handle joins */
20:     for all e = (a, a') in E_MQ do
21:         add to WHERE the predicate "cond(e)" with corresponding variables
22:     end for
23:     optimize WHERE condition
24:     add to WHERE for each pair a, a' ∈ A_MQ a predicate "$a.alias + "_ext" !=
    $a'.alias + "_ext""
25:     combine FOR, LET, WHERE, RETURN to cquery and return
26: end function
```

All-$size_{max}$ Semantics

In the first case, we want to explore the complete data. Thus, we do only restrict the maximum size of valid object networks, and in consequence, the maximum size of materialization queries. The user wants to get all information. This kind of queries allows the possible detection of new relationships or information during a post processing. We denote this type of queries as All-$size_{max}$ keyword queries. For example, using a maximum size of $size_{max} = 1$ would return single objects as results without additional information. The corresponding problem definition is formally stated as follows.

Definition 5.9 All-$size_{max}$ query results.
Given are a keyword query Q, a virtual document D, and a maximum size $size_{max}$. The All-$size_{max}$ result with respect to Q and D is a list of valid result object networks that are the result of valid materialization queries according to Q, D, and $size_{max}$. □

```
FOR $a1 IN concept[name="Painting"],
    $a2 IN concept[name="Painter"],
    $a3 IN concept[name="Museum"]
LET $a1_ext := extension($a1),
    $a2_ext := extension($a2),
    $a3_ext := extension($a3)
WHERE $a2_ext/Name ~= "rubens" and $a2_ext/City ~= "prague"
      and $a2_ext/Name = $a1_ext/Artist and
      $a1_ext/Institution=$a3_ext/Name and $a1_ext!=$a2_ext
      and $a1_ext!=$a3_ext and $a3_ext!=$a2_ext
RETURN <result>$a1_ext, $a2_ext, $a3_ext</result>
```

Figure 5.7.: Listing materialization query translation result

Top-k Semantics

All-$size_{max}$ queries may create a large set of data that is hard to explore and expensive to create. Top-k queries are another approach. Top-k queries retrieve the k best scoring materialization queries according to a keyword query Q that have a non-empty result set. That means that the result can have more than k object networks because a materialization query can return more than one object network. This type of queries is helpful in situations where the user wants to have the best results, but she does not have expectations about the structure or size of the results. For example, consider the query “fine arts Flemish Prague”. Results can include only cultural assets, but also networks containing institutions and artists as well as FineArts objects. The problem is stated as follows.

Definition 5.10 Top-k query results.
Given a keyword query Q, return valid result object networks that are the result of the k highest scoring valid, non-empty materialization queries mq according to a virtual document D and scoring function $\text{score}(mq, Q)$. *The answer list has to be sorted in descending score order.* □

Concept Network Semantics

Another application of keyword queries is the further exploration of the integrated data. In this case, the user formulates a keyword query and wants to discover the different types of connections between the keywords in the form of object networks. Materialization queries and object networks conform to concept networks. Many different materialization queries and object networks might have the same concept network. We say, the type of a connection between keywords is defined by the concept network. For example, the user formulates a query “Holbein book paintings.” The result may comprise object networks containing only book objects, networks between painter and paintings or authors and books, etc. The user can inspect the instances of all types.

Definition 5.11 All concept network query result
A list of object networks ON will be an all concept network query result with respect to query Q, a scoring function score and a virtual document D, if

- *all object networks in the list ON are valid answers with respect to Q and $size_{max}$,*
- *the list contains all distinct types (concept networks) of answer object networks with maximum size $size_{max}$, and*
- *for each distinct concept network in the answer list, the list contains the top-k object networks according to function score*

□

Another variant of concept network semantics is to obtain the best n concept networks and their corresponding k best instances.

5.3.4. Materialization Query Scoring Function

The ranking function of the query model is based on the score of materialization queries. That means that we score queries but not object networks. The advantage of this approach is that we can score a query based on the information in the virtual document. In addition, the ranking function is independent of different, local scoring functions. The disadvantage of the approach is that the result sets of object networks have the same score and many object networks cannot be distinguished by score. However, we assume that the result sets of many queries are small, which mitigates the disadvantage.

The scoring function of materialization queries is based on the structure of the query as well as on term weights in the bags of words of the virtual document. We describe the content score with the help of the weight of term assignments based on the term weight in the corresponding bags. The term weights are based on the $tf \cdot idf$ scoring schema [BYRN99]. Here, we distinguish between concept and data level scores. At last, we will show that the scoring function is monotonic according to scores of single nodes. In the discussion below, we assume a virtual document $D = (N_D, E_D, V)$.

Concept level term weights

Terms on concept level occur either in the terms of concept nodes or in the term bag of properties. They represent the description of concepts and properties. In order to compute the term weight, we use the term frequency (tf) and inverse document frequency (idf). The term frequency describes the importance of a term in one term bag. Global term bags contain terms from global labels as well as from mapping information, e.g., local element names or filter values. We say a term is more important if it covers more sources. That means, a term from a global label that covers all sources is more important than a term originating from one source mapping representing only one source. Let $S(i)$ be the set of sources mapped to a concept i or property i. Let

$S(i,t)$ be the sources covered by a term $t \in i.terms$ for a concept i or a property i. Then, we formalize the term frequency $tf(t, i.terms)$ as

$$tf(t,i) = \begin{cases} \frac{|S(i,t)|}{|S(i)|} & \text{if } S(i) \neq \emptyset \\ 1 & \text{if } S(i) = \emptyset. \end{cases} \tag{5.1}$$

The second case is used for concept properties, i.e., edges in the virtual document. Edges are not provided by sources, therefore, all terms are global terms.

The second component of the term weights is the inverse document frequency *idf*. We distinguish between *idf* values for each type of elements. We have on concept level the values $idf_C(t)$ and $idf_P(t)$ for a term t. The first describes the inverse document frequency for concept nodes and the second is the inverse document frequency for properties. Let d_C the number of concept nodes, and d_P the number of properties, i.e., $d_C = |N_D|$ and $d_P = \sum_{n \in N_D} |n.properties|$. The inverse document frequencies are defined as

$$idf_C(t) = \ln \frac{d_C + 1}{d_C(t)} \tag{5.2}$$

$$idf_P(t) = \ln \frac{d_P + 1}{d_P(t)} \tag{5.3}$$

with $d_C(t)$ the number of concept nodes (d_P the number of properties) containing the term t following the approach in [Sin01]. Combining the tf and *idf* values of terms, we obtain the weight of a term in a concept node term bag or in a property term bag as

$$w_C(t,n) = tf(t,n) \cdot idf_C(t) \tag{5.4}$$

$$w_P(t,p) = tf(t,p) \cdot idf_P(t) \tag{5.5}$$

with $n \in N_D$ a concept node of a virtual document and $p \in n.properties$ a property of a concept node $n \in N_D$.

Data level term weights

The virtual document represents the extension of concepts as bags of terms. For every concept node, it contains one bag of terms for every pair of source and property. We compute for each bag the term weight for the contained terms. Subsequently, the term weight combines the single weights to a term weight for every property of one concept. In the first step, we consider categories as simple terms.

Assume a concept node n and the property $p \in n.properties$. The property p has a set $p.sources$ of pairs $(s, T(s,p))$. The term frequency of a term i in term bag $T(s,p)$ is the number of occurrences $occ(t, T(s,p))$. We assume this number is the number of objects in the extension concept n that contain the term t for the given property p

and come from the given source s. We normalize the term frequency according by the relative bag size. That means, the term frequency is

$$tf(t, T(s,p)) = \frac{1 + ln(occ(t, T(s,p)))}{\frac{|T(s,p)|}{avgSize}} \tag{5.6}$$

with $avgSize$ the average size of all source bags in the virtual document. This is a simplified computation from [Sin01].

The inverse document frequency of a term t is again the number of properties that contain the term in at least one of the source term bags. We denote the number of such properties as $d_S(t)$. Thus, we define the idf_S value as

$$idf_S(t) = \ln \frac{d_P + 1}{d_S(t)}. \tag{5.7}$$

Combining the tf values and the idf value, the following Equation (5.8) computes the weight of a term t for the extension of a property p :

$$w_S(t,p) = \left(\sum_{(s,T(s,p)) \in p.sources} tf(t, T(s,p)) \right) \cdot idf_s(t). \tag{5.8}$$

Scoring function

The scoring function score(mq, Q) consists of the score of the content and the score of the structure. Assume $mq = (A_{mq}, E_{mq}, \mathbf{I})$ is a valid materialization query according to keyword query Q. We assume, the weight of a term in the query is denoted as $w_Q(t, Q)$. In general, the query weight is 1, but we show the use of the query weight for query expansion in Section 5.4. First, we consider the term weights and get the content score

$$\begin{aligned} \text{contentscore}(mq, Q) = w_1 &\cdot \text{schemascore}(mq, Q) \\ &+ w_2 \cdot \text{datascore}(mq, Q) \end{aligned} \tag{5.9}$$

with w_1 and w_2 be two weights with $w_1 + w_2 = 1$. The schema score is the sum of all assignments to concepts and properties. We define the schema score as

$$\begin{aligned} \text{schemascore}(mq, Q) = &\sum_{\substack{n \in A_{mq}, (t, uri, true) \in \text{assign}(n), \\ t \in Q}} w_C(t,n) \cdot w_Q(t,Q) \\ &+ \sum_{\substack{n \in A_{mq}, (t, uri, \text{exists}(p)) \in \text{assign}(n), \\ t \in Q}} w_P(t,p) \cdot w_Q(t,Q) \\ &+ \sum_{\substack{e \in E_{mq}, (t, uri, true) \in \text{assign}(e), \\ t \in Q}} w_P(t, e.label) \cdot w_Q(t,Q) \end{aligned} \tag{5.10}$$

The equation summarizes the weights of terms in the query that are interpreted as schema level terms. The first term summarizes the concept assignments, i.e., the respective term weights in the concept term bags, the second term computes the sum of all property assignments to concept and literal properties, and the third term builds the sum of all assignments to concept properties. We assume that all terms t occur in Q either as plain query terms or concept or property labels in concept-based query terms.

The data score is computed as the sum of weights of terms that occur in the query Q and in data level assignments of the materialization query. First, we provide a preliminary datascore(mq, Q) function:

$$\text{datascore}(mq, Q) = \sum_{\substack{n \in A_{mq}, (t, uri, pred) \in \text{assign}(n), \\ t \in Q}} w_S(t, p) \cdot w_Q(t, Q)$$

with $pred$ be either $p = t$ or $p \sim= t$. A data level term t may occur in different sources denoted as $S(t)$. Two data level terms assigned to one concept node might occur in different source sets. Thus, the objects have to be combined from different sources to obtain valid results. We argue, in this case the score has to be lower. Thus, we introduce the compactness of data term assignments assign(n) to a node $n \in A_{mq}$. It is defined as

$$\text{compact}(n) = \frac{|\bigcap_{(t, uri, pred) \in \text{assign}(n)} S_t| + 1}{|\bigcup_{(t, uri, pred) \in \text{assign}(n)} S_t| + 1}. \tag{5.11}$$

The compactness describes the ratio between common sources and supported sources. Without common sources, the score will be minimal. We modify the datascore(mq, Q) as follows to obtain the final definition:

$$\text{datascore}(mq, Q) = \sum_{n \in A_{mq}} \left(\text{compact}(n) \cdot \sum_{\substack{(t, uri, pred) \in \text{assign}(n), \\ t \in Q}} w_S(t, p) \cdot w_Q(t, Q) \right) \tag{5.12}$$

The final component of the scoring function is the size of the materialization query. We use as size the number of concept aliases in the query, i.e., size$(mq) = |A_{mq}|$. Furthermore, a more compact query is a more relevant result. Thus, we combine all components of the scoring function to the final function presented in Equation (5.13).

$$\text{score}(mq, Q) = \frac{\text{contentscore}(mq, Q)}{\text{size}(mq)} \tag{5.13}$$

The scoring function comprises the importance of term assignments as well as the structure of the query. It is used as proof of concepts and demonstrates all relevant parts but it is not optimized for effectiveness. However, it has the important property of monotonicity [HGP03] allowing the efficient computation of results. The monotonicity is according the partial scores of the concept aliases of the query. Assume two materialization queries mq_1 and mq_2 of the same size. Furthermore, all but one node have the same set of assignments, thus, the same score. For the remaining node n_{1j}

and n_{2j}, we assume that n_{1j} has assignments with a higher score than n_{2j}. From that, it follows that score(mq_1, Q) > score(mq_2, Q). The function has this feature, because the schemascore(mq, Q) and datascore(mq, Q) are summarizations of the scores of the aliases and edges. Thus, we have a monotonic ranking function and can use similar techniques as proposed by Hristidis et al. [HGP03].

5.4. Query Expansion

Concept-based query terms help to restrict the result sets to relevant object networks, because the user can give hints in which extensions the system should search for value keywords. However, the user can overspecify the query in that way. We propose two ways of query expansion to combine the exactness of a concept-based keyword query with the flexibility of a general keyword query. The first query expansion is the usage of semantically related concepts. A second query expansion reuses category hierarchies to rewrite value keywords.

5.4.1. Semantic Distance of Classes

The global concept schema describes the relationships between concepts and categories. We now define what semantically related classes are and how close they are related.

Semantically related classes

Classes are organized by subClassOf relationships (as in YACOB, we will denote it as **is_a**) and create hierarchies of classes in that way. A class hierarchy has one distinct class, the root class. A root class is either a concept or a category that does not have a super-concept or a super-category, respectively. The hierarchy contains all classes that are directly or indirectly connected to the root class by the inverse **is_a** relationship.

Definition 5.12 Class hierarchy and Semantic Relationship
A class hierarchy is a tuple $H = (C, root, \textbf{is_a})$ *with* C *a set of classes and* $root \in C$ *the root node. For every class* $c \in C, c \neq root$ *exists exactly one class* $c' \in C$ *with* $\textbf{is_a}(c, c')$. *We say two classes are* semantically related *if they belong to the same hierarchy* H, *denoted as* semRelated(c, c'). □

Example 5.4 *Figure 5.2 on page 87 contains three concept hierarchies: a hierarchy of person related concepts with concept "person" as root, the cultural asset hierarchy, and the institution hierarchy. For example, the concepts "print" and "sculpture" are semantically related because they belong to one hierarchy.*

Semantic distance

The *semantic distance* between two classes describes how closely related two classes are. For computation of the distance, we use the structure of the class hierarchies. The distance definition is based on the following assumptions:

1. A concept has a shallow extension and a deep extension. The deep extension also comprises the extensions of descendants. A category comprises also terms of sub-categories. We assume, a user wants to have all objects or all categories that belong to the deep extension. Hence, we assume that the descendants of a class should have the same score as the class.

2. Siblings are closely related but less than descendants.

3. The distance depends on the degree of specialization. The siblings of a more specialized class are closer than those of a more general class. For example, the concepts etching and engraving in Figure 5.2 are more closely related than the concepts sculpture and graphics.

We now develop a distance measure that reflects these intuitive assumptions. First, we define the degree of specialization of a class. The level of a class c in a hierarchy H is denoted as $\mathrm{level}(c, H)$. The level is defined as the number of nodes in the path from the root class *root* to class c. Hence, the root has the level 1. The height $\mathrm{height}(H)$ of a hierarchy H is defined as the maximum level of any class in the hierarchy, i.e., $\mathrm{height}(H) = \max_{c \in H} \mathrm{level}(c, H)$. Now the degree of specification of a class c in the hierarchy H can be computed as the ratio between level and height: $\frac{\mathrm{level}(c,H)}{\mathrm{height}(H)}$. We illustrate the values in Figure 5.8. The hierarchy has a height 4, the root node has the level 1 and class e (black node) has the level 3, for example. The *lowest common ancestor* $lca(c, c')$ of two classes c and c' is that node, that is an ancestor of c and of c', and it does not exist another node x, which is an ancestor of c and of c' and a sub-class of $lca(c, c')$. For example in Figure 5.8, node b is the lowest common ancestor of class d and h.

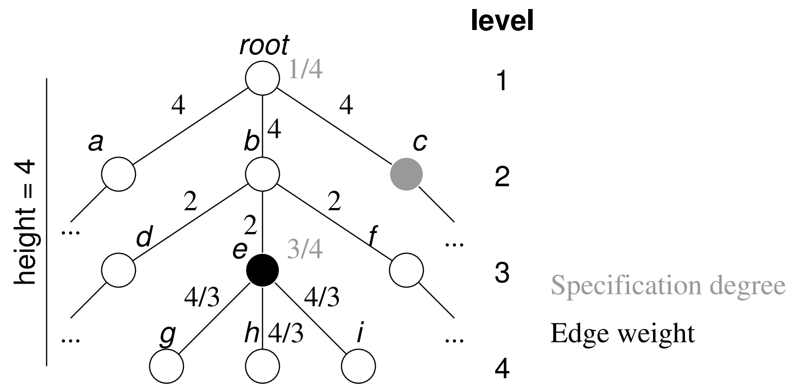


Figure 5.8.: Illustration of Semantic Distance

We assign to every subClassOf edge in the hierarchy a weight. The weight of the edge equals to the inverse specialization degree of the parent, i.e.,

$$\forall (c, c') \in \mathbf{is_a}(H) : w_{cp}((c, c')) = \frac{\mathrm{height}(H)}{\mathrm{level}(c', H)}. \tag{5.14}$$

That means that the edges from a parent to its children have all the same weight and the edges closer to the root have a higher weight. For example, in Figure 5.8 the edges

from root to its children have the weight of $4 = 4/1 = \frac{\text{height}(H)}{\text{level}(root,H)}$. The weight of the edges of e to its children is $\frac{4}{3}$. Let c be a descendant of a class c'. The weight of the path along the subClassOf axis between c and c' is defined as

$$w_{da}((c,c')) = \text{height}(H) \cdot \left(\sum_{\text{level}(c',H) \leq i < \text{level}(c,H)} \frac{1}{i} \right). \tag{5.15}$$

The equation is deduced from the sum of the weights w_{cp} in the path between both classes. For example, the path between the descendant-ancestor pair $(e, root)$ has the weight $w_{da}(e, root) = w_{cp}(e,b) + w_{cp}(b, root) = \frac{4}{2} + \frac{4}{1} = 4 \cdot (\frac{1}{2} + \frac{1}{1}) = 6$ with $\text{height}(H) = 4$ and $\text{level}(e, H) = 3$ and $\text{level}(root, H) = 1$.

In order to meet the assumptions of the beginning of this section, we define the *semantic distance* between two classes $c, c' \in H$ in a hierarchy H as

$$\text{semDist}(c, c') = w_{da}(c, lca(c, c')) \tag{5.16}$$

with $lca(c, c')$ the lowest common ancestor of c and c'. The semantic distance definition of Equation (5.16) has the following implications:

1. It holds $\text{semDist}(c, c') = \text{semDist}(c', c) = 0$ if $c = c'$.

2. It is not a metric because $\text{semDist}(c, c')$ is not the same as $\text{semDist}(c', c)$ in every case. For example, in Figure 5.8 the distance between node e to node c is $\text{semDist}(e, c) = 6$ but between c and e is $\text{semDist}(c, e) = 4$. This is intuitive in our scenario, because e is a specialized class, any extension to higher classes will add many new results. On the other hand, if we search for a class, we will expect also results of the sub-classes.

3. If class c is a descendant of class c', the semantic distance between c' and c is zero, because the $lca(c', c) = c'$ and $w_{da}(c', c') = 0$. That follows the intuition that descendants should be included.

4. The inequality $\text{semDist}(c, c') \leq \text{semDist}(c, c'') + \text{semDist}(c'', c')$ holds. The distance is expressed as the distance to the least common ancestor $w_{da}(c, lca(c, c'))$. If c' is a descendant of c'', then it is $\text{semDist}(c, c') = \text{semDist}(c, c'') + \text{semDist}(c'', c')$ because, in this case, the $lca(c, c'') = lca(c, c')$ and the distance $\text{semDist}(c'', c')$ is zero. If c'' is not a subclass of c', the $lca(c, c')$ is always a super-class or equal to the $lca(c', c'')$. In consequence, the inequality holds.

Now we describe how to use the semantic distance for classes in a hierarchy for query expansion of concept and categories, respectively.

5.4.2. Concept Expansion

The expansion of concepts is used in concept-based keyword queries. A concept-based query term is a triple $t = (painting : title : scene)$. Now we assume that we find a valid keyword assignment $(painting, painting, true)$ for the concept

painting. Furthermore, we assign the property keyword $(title, drawing, exists(p))$ and $(scene, drawing, title \sim= "scene")$ to a concept *drawing*. An assignment of t to a concept is only valid, if all three terms are assigned the same concept, which is not the case. That means that a user has disadvantages because of her query hint using concept-based query terms and does not get any results.

To solve that problem, we expand the concept to semantically related concepts. The concepts “painting” and “drawing” are semantically related. We modify the interpretation $(painting, painting, true)$ to $(painting, drawing, true)$. In order to reflect the semantic distance, we calculate the corresponding weight $w_C(painting, drawing)$ using the semantic distance. Assume to semantically related concepts c and c' and term t, the score modification is

$$w_C(t, c') = \frac{1}{1 + \text{semDist}(c, c')} \cdot w_C(t, c). \tag{5.17}$$

Example 5.5 *Consider the concept schema in Figure 5.2 on page 87 and a concept-based keyword query $Q = (engraving : artist : holbein)$. Given the exemplary extensions in Figure 5.3 on page 87, the query would return an empty set. However, we can extend the query using concept expansion and use a new query $(print : artist : holbein)$. “Print” is the parent concept of engraving, and their distance is $\frac{5}{4}$. In consequence, the query term weight of print is $\frac{1}{1+\frac{5}{4}}$. The modified query retrieves the object pr_1 in Figure 5.3.*

5.4.3. Category Expansion

Searching for categories using keyword queries is a two step process. Given are a virtual document $D = (N_D, E_D, V)$ with $V = (K_V, E_V)$ a number of category hierarchies and the original keyword query Q_{orig}. Let $t \in Q_{orig}$ refer to a category. In the first step, we find all categories that contain t, i.e., $V_t = \{v | v \in K_V : t \in v.terms\}$. From the category set V_t, we create a set of keyword queries $\mathbf{Q}(V(t))$. In every query $Q \in \mathbf{Q}(V(t))$, we replace t by a category $v \in V_t$. Now we search for valid materialization queries for every query $Q \in \mathbf{Q}(V(t))$.

However, in order to exploit the subClassOf relationships as well as the semantical relationship between categories, we can expand $\mathbf{Q}(V(t))$ further to $\mathbf{Q}^*(V(t))$ using semantically related categories with respect to elements of V_t. For every query $Q \in \mathbf{Q}(V(t))$, we create a query Q' by replacing the category $v \in V_t$ by a semantically related category v' and set the corresponding query weight to

$$\begin{aligned} w_Q(v', Q') &= \frac{1}{\text{semDist}(v, v')} \cdot w_Q(v, Q) \\ &= \frac{1}{\text{semDist}(v, v')} \cdot w_Q(t, Q_{orig}). \end{aligned} \tag{5.18}$$

Valid materialization queries to $\mathbf{Q}^*(V(t))$ create results to expanded queries of Q_{orig} with the corresponding lower ranking.

Example 5.6 *We illustrate a hierarchy of picture subjects in Figure 5.9. For example, consider the concept-based keyword query $Q = \{painting, motif, landscape\}$. The*

value term landscape can be found in the category "landscape". Now the system extends the query Q *using category extensions. For example, a query* $Q' \in \mathbf{Q}^*$ *can be* $Q' = \{painting, motif, "seascape"\}$. *The category "seascape" has the same score as the original term. A further query can be* $Q'' = \{painting, motif, "StillLife"\}$. *The query term weight of category "Still Life" is* $w_Q("StillLife", Q'') = \frac{1}{1+\frac{1}{3}} * w_Q(landscape, Q)$.

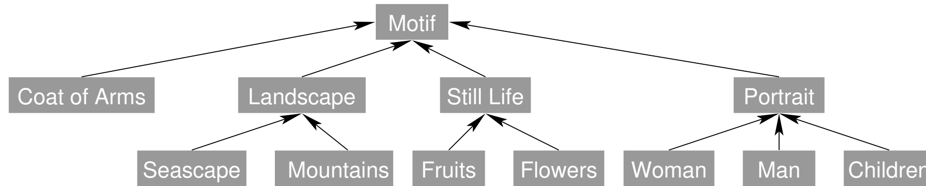


Figure 5.9.: Category Expansion Example

5.5. Related Work

We compare the proposed approach by using the categories defined in Section 4.2: query language, data model and result types, ranking methods, and evaluation type.

Query language

The query language is labeled keyword queries. A label consists of two parts: concept and property label. Thus, the proposed approach belongs to the group of labeled keyword queries [CMKS03, HN02]. However, we utilize the semantic model for query expansion similar to XXL [TW02a]. The difference is that we exploit the Yacob model directly without using auxiliary data sources, and we support categorical data values, too.

Data model and result type

The data model is based on the concept-based model of Yacob. It is based on two concepts: object networks and materialization queries. Object networks are connected global objects. They are equivalent to minimal total joining tuple trees as defined in Hristidis and Papakonstantinou [HP02]. Materialization queries are used to create object networks. Materialization queries are similar to candidate networks in relational databases. The difference is that materialization queries specify in which property a term occurs and allows edge keywords. The queries are adapted to the concept-based data model. Materialization queries also relate to the concepts of Demidova et al. [DZN10] and SQAK [TL08]. In both works also queries are constructed as interpretations from keyword queries. Demidova et al. use the relational data model without concept hierarchies. In the SQAK system, a subset of SQL is created including grouping and aggregation. Calado et al. [CdSV$^+$02] and the EasyQuerier system [LMM07] map keyword queries to query forms for Deep Web search. In this case, populated forms are materialization queries and the results are objects or HTML pages. However, both works do not discuss information units consisting of different objects. Liu

et al. [LLY+05] describe how natural language queries select structured databases by computing the semantic distance between a query graph and a schema graph. That means that only the schema keywords are addressed in the approach.

We proposed three meaningful query semantics: all results, top-k results, and top-k concept networks. The systems Discovery [HP02] and DBExplorer [ACD02] proposed all results, too. Most keyword search systems consider a top-k approach. The top-k concept network semantics groups materialization queries and their results. It orders the results by the highest ranked non-empty materialization query. Similar grouping was proposed by Hristidis et al. [BHK+03] or in the form-based search [CBC+09]. The first approach does not consider ranking, and the second allows SQL as form language and not a concept-based join query.

Ranking function

We use a monotonic ranking function based on the $tf \cdot idf$ framework to rank materialization queries with respect to keyword queries. We distinguish schema and data scores. Query expansion manipulates the query and schema term weights to rank query expanded queries. The term weight manipulation for query expansion is similar to the XXL [TW02a] approach but follows a different intuition. XXL uses the semantic distance based on the Word Net ontology. In contrast, we try to rank the user's assumption that objects of sub-concepts are included in the result.

Ranking of queries instead of object networks is related to the database selection of unstructured [CLC95, GGMT94] and structured sources [YLST07, VOPT08]. The ranking function of materialization queries is similar to the candidate network ranking in Xu et al. [XIG09]. Xu et al. use the CN ranking for improving the ranking of joining tuple trees. Demidova et al. [DZN10] used a probabilistic formula to rank queries constructed from a keyword query. The formula combines document frequency of terms in an attribute and the probability of a query template (equivalent to a concept network). The authors also consider schema and value keyword occurrences but assume given query templates from a query log. They assume the relational model and do not consider query expansion.

The proposed ranking function is monotonic. That means that it is possible to adapt tuple set algorithms [HGP03, SLDG07]. Sayyadian et al. propose different weights for foreign key relationships [SLDG07]. These weights are based on the quality of approximate foreign key relationships and not on matching named concept relationships. In contrast, Liu et al. [LYMC06] and the SPARK system [LLWZ07, Luo09] propose non-monotonic ranking functions, which require specific evaluation methods. Further approaches are inspired by the PageRank method like XRank [GSBS03] or BANKS [HN02]. Because of the integration setting, we consider global and local schema information for ranking. Similar approaches are EasyQuerier [LMM07] and SINGAPORE [DD01b]. EasyQuerier maps keyword queries to an integrated Web database interface. The mapping comprises two steps: first domain mappings and second attribute mappings. The domain mapping tries to find the right domain for the given keyword; the second step finds matching attributes of an integrated query form. The ranking is based on a weighted function which combines the similarity of the query to a domain and the attributes. The system SINGAPORE is in some

points similar to our approach, because local and global schema information are represented in a global data tree. However, the SINGAPORE system does not try to define integrated extension sets and semantic connection between extensions.

Evaluation method

The evaluation method is not discussed in this chapter. The data is not materialized on the global level, and we do not build a data graph of global objects. Instead, we interpret keyword queries as concept-based join queries. The join queries are executed to obtain the actual object networks as results. Thus, the approach is schema graph-based.

5.6. Summary

We defined concept-based keyword queries on a concept-based data model provided by the YACOB system. Inherently, we considered that the data is not materialized but virtually integrated from different sites. We defined the virtual document as the foundation of the keyword search. A key point is that all information is used to describe concepts and properties. That includes source descriptions and mappings. We presented concept-based keyword queries and their semantics. The query answers are lists of object networks. We argued that object networks are generated by materialization queries. These queries are interpretations of the keywords. We proposed a ranking function that includes the schema scoring and data scoring parts as well as considers the structure. It is a monotonic ranking function. The query expansion using class hierarchies can make concept-based keyword queries more usable. For this, we defined semantically relatedness and a distance function. The function is not a metric but follows the intuition for concept-based materialization functions. The chapter extended previous results [GDSS03, Dec04, Gei04, SGS05] in the following points:

- Object networks are supported as results instead of objects of single concepts.
- Labeled keyword queries are supported additionally to plain keyword queries,
- A ranking function of object networks and materialization queries has been defined.
- Query expansion mechanisms extend the previous approaches that included only subclasses and not semantically related classes.

In the following two chapters, we discuss how to process concept-based keyword queries using the schema graph-based method.

6. Concept Query Generation

We defined concept-based keyword queries and their semantics over virtually integrated databases in the previous Chapter 5. This chapter deals with the efficient execution of keyword queries using a schema graph-based approach. The keyword query processing consists of two main steps: concept query generation and their execution. The first step translates keyword queries into materialization queries, also denoted as concept queries. These queries are used to create object networks in the second step. This chapter deals with the efficient concept query generation (see Figure 6.1), while Chapter 7 investigates efficient concept query processing. This chapter extends the keyword search of the YACOB system [Dec04, GDSS03, Gei04] in the following points. The keyword index is extended by using Dewey identifiers [Dew04] for hierarchy information and single concept queries are created for keyword query subsets instead of only for complete keyword queries. Finally, object networks instead of objects are supported as results with respect to a keyword query.

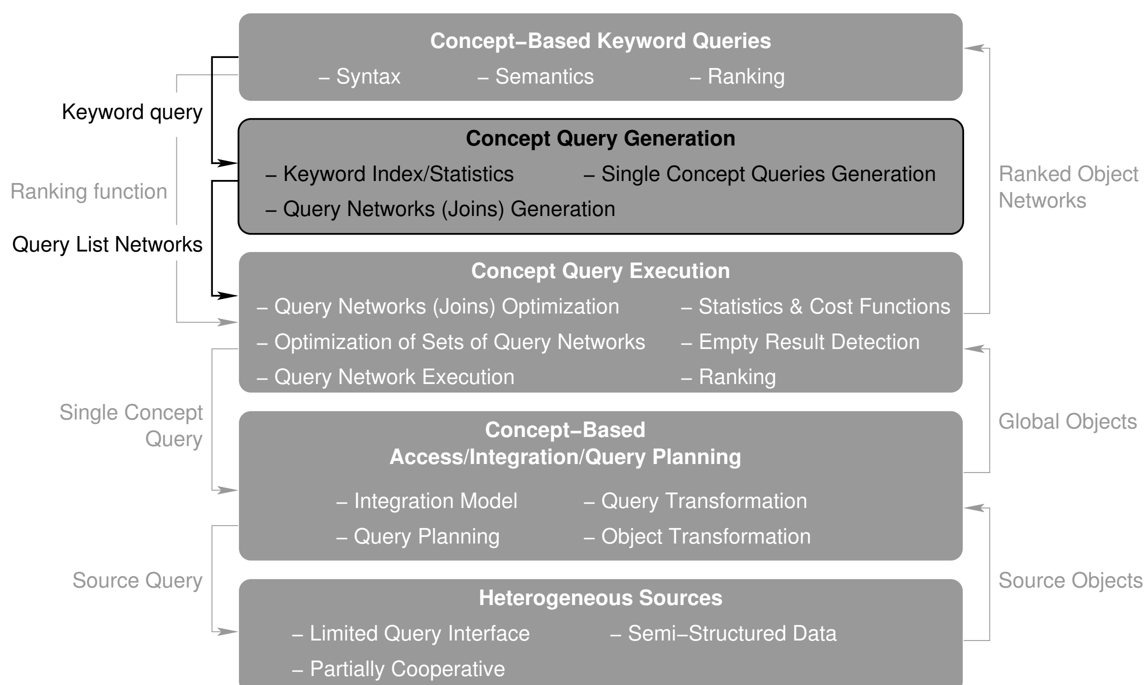


Figure 6.1.: Overview: concept query generation

We outline the complete keyword search process in Section 6.1 including concept query generation and concept query processing steps. We describe the keyword index and the generation of single concept queries in Section 6.2 and 6.3, respectively. Based on lists of single concept query, we develop an efficient enumeration of query list

networks in Section 6.4. Query list networks represent a set of concept queries and are the input of the concept query processing described in the following Chapter 7. We conclude the chapter with the discussion of related work in Section 6.5 and a summary in Section 6.6.

6.1. Overview

Figure 6.2 illustrates the complete keyword query process. The approach belongs to the class of schema graph-based keyword search systems reviewed in Chapter 4. At the

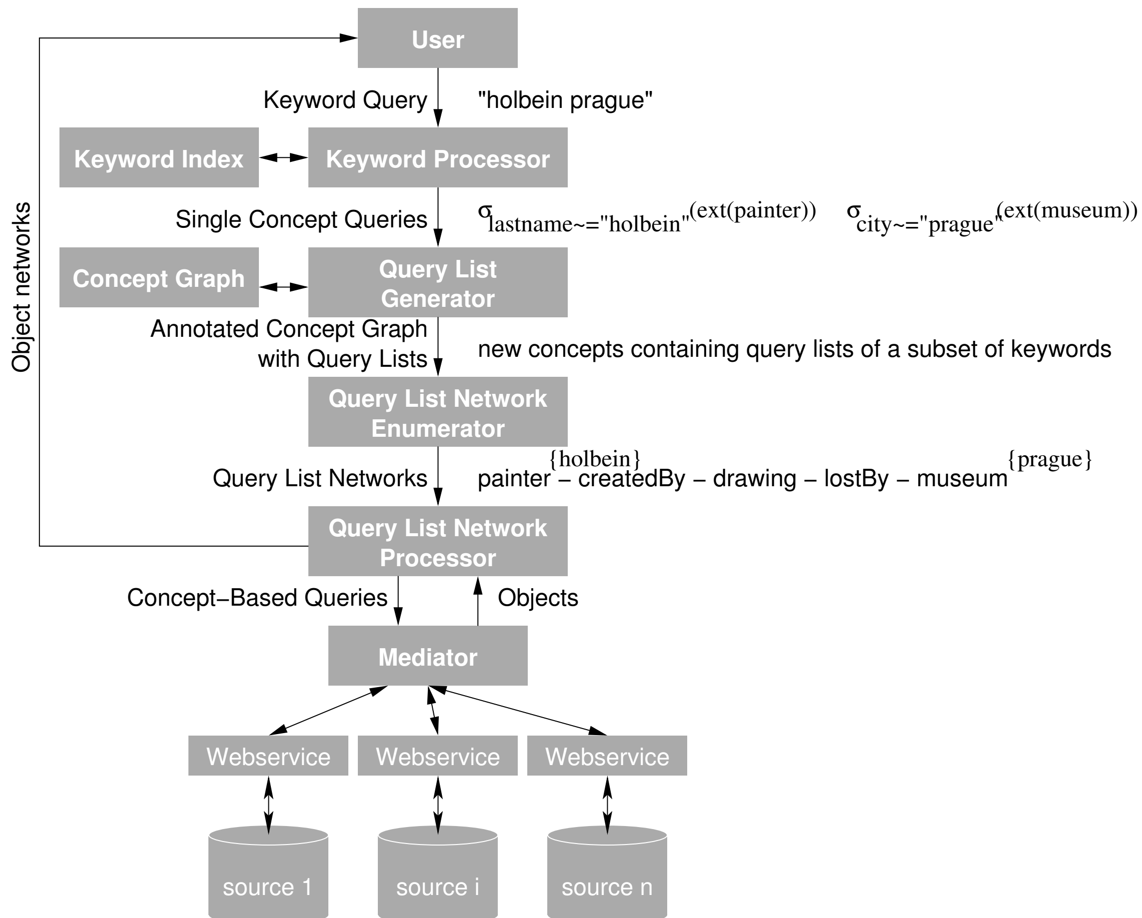


Figure 6.2.: Keyword query processing steps

beginning, the user formulates a keyword query. An exemplary plain keyword query is "holbein prague". The query is sent to the *Keyword Processor*. The processor parses the query and sends the query terms to the keyword index. From the returned index entries, the processor creates concept queries. A single concept query is a concept-based, structured query over one concept extension. This is one difference to the related systems because we deal with query lists instead of tuple sets. A single concept query is an interpretation of the complete keyword query or a subset of it. In a later step, the single concept queries are combined to complete materialization queries that contain all keywords. Hence, we assume AND semantics in this thesis.

The *Query List Generator* component uses the single concept queries and creates query lists. A query list is a list of single concept queries that refer to the same concept or edge and contain the same subset of keywords. The support of named edges is a further extension to related schema graph-based keyword search systems. The query list generator creates all possible query lists from its input. For every query list, we create a new concept graph node (or edge) representing the corresponding list. During that process, the query list generator extends the concept graph and finally creates the annotated concept graph. The annotated concept graph comprises the original concepts and edges as well as query list concepts and query list edges. Figure 6.3 shows an exemplary concept graph containing three concept list concepts and one query list edge.

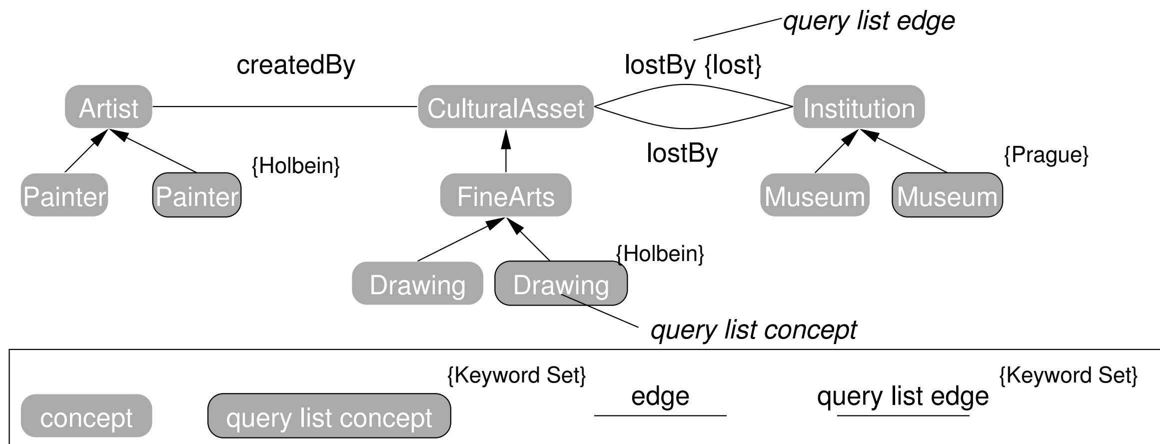


Figure 6.3.: Exemplary annotated concept graph

The annotated concept graph is the input of the *Query List Network Enumerator*. The enumerator creates networks, denoted as query list networks, of query list concepts connected via free concepts and concept properties. Assuming AND semantics, every query list network contains all keywords. A query list network (QLN) represents a set of materialization queries that are used for retrieval of objects and to construct object networks. Figure 6.4 illustrates two query list networks for the exemplary query $\{prague, holbein\}$. `QLN_1` represents queries containing drawings and museums, and `QLN_2` represents artists connected to museums through created and lost cultural assets.

The output of the query list enumerator is the input of the *Query List Network Processor*. The QLN processor creates the actual materialization queries and executes them. The component sends single concept queries to the YACOB mediator. The mediator component executes the queries and returns sets of objects. The object sets are combined to sets of object networks by the QLN processor. The combination of object sets might induce new mediator queries by using bind join operations [GMPQ+97, GW00]. In that way, the QLN Processor creates step by step object networks. The system returns the object networks to the user.

In the remaining chapter, we discuss the steps including the generation of query list networks. We start with the description of the keyword index and process in the following section.

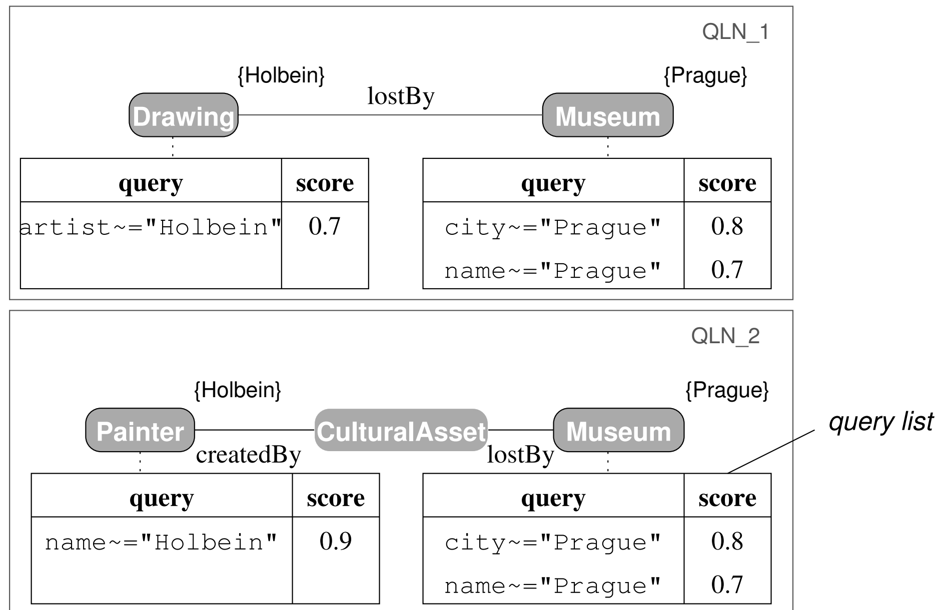


Figure 6.4.: Exemplary query list networks

6.2. Keyword Processor

The keyword processor has the task to find interpretations of a given query terms, i.e., a concept or property name, or a category selection or a keyword containment selection. The result of the keyword processing step is a list of tuples consisting of keyword index entries. A keyword index entry comprises all information to determine the position of a keyword occurrence and with that, the interpretation of the keyword. Based on the result of the keyword processor, the next step creates single concept queries. The results of this section are partially based on [Dec04, GDSS03, SGS05].

6.2.1. Keyword Index

The *keyword index* stores all information about keyword occurrences in a virtual document $D = (N_D, E_D, V)$. The index comprises the concept level terms as well as the terms on data level. We start the discussion of the keyword index with the description of the indexing process. Figure 6.5 illustrates the indexing process that is the offline phase of the keyword search system. In order to represent the data level, we extract from each source keyword statistics. For example, the statistics comprise how often (in how many objects) a keyword occurs in the extension provided by a source. The statistics are either extracted using a *crawler* through the mediator system, or the sources provide the statistics directly via a protocol, e.g., [GCGMP97, IBG02]. These statistics are denoted as *source content descriptions* and form the first input of the indexer. The second input is the concept schema and the mapping information. Terms of the concept schema as well as local element names from the mappings are extracted and added as schema level terms to the *keyword index*. The *indexer* combines the schema level terms with source content descriptions to the final keyword index.

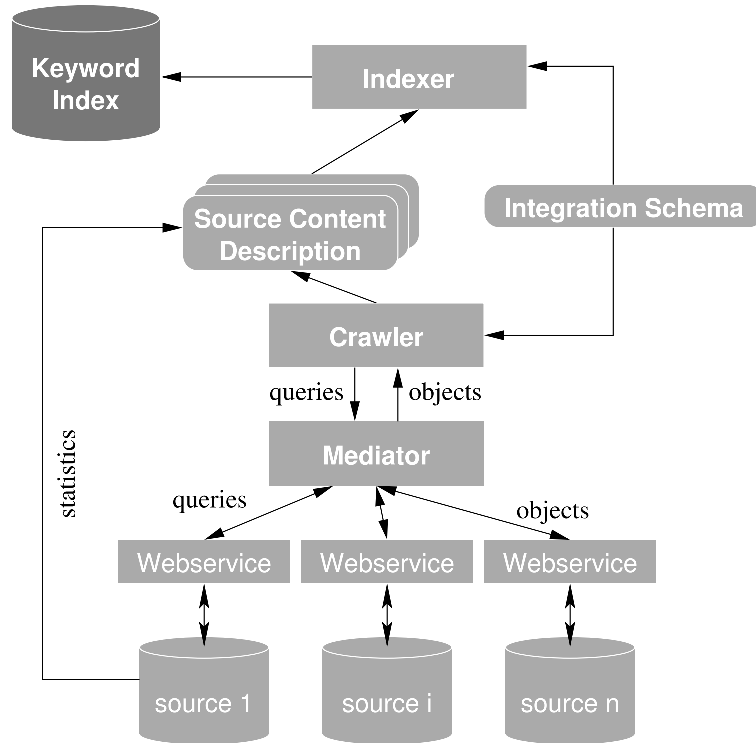


Figure 6.5.: Offline phase of the keyword search system

Source content description

Source content descriptions provide statistics of keywords in the concept extensions. In order to compute the term weights as described in Chapter 5, the content description of a source $s \in Sources$ comprises the following information. First, it contains the size (number of objects) provided of the source s for a concept c, i.e., the value $size(c, s)$. Second, the content descriptions contain the document frequency of a keyword t for a concept c and property p in the source s. For this, we retrieve for every term $t \in \mathcal{T}$ in a source s a list of document frequencies. The document frequency $df(t, c, p, s)$ is the number of objects of concept $c \in \mathcal{C}$ containing the term t in a data property $p \in \mathcal{P}$. Alternatively, the descriptions are delivered using the local element names with the advantage of independence from the global level. As we do not need the complete source content in the regular case, we use the concept schema as structure.

In summary, the source content description $SCD(s) = (Sizes(s), DF(s))$ of a source s consists of two relations $Sizes(s)$ and $DF(s)$. The relation $Sizes(s)$ contains the concept extension size information and is defined as $Sizes(s) \subset \mathcal{C} \times \{s\} \times \mathbb{N}$. The relation DF contains all document frequencies of the terms in the source s, i.e., $DF(s) \subset \mathcal{T} \times \mathcal{C} \times \mathcal{P} \times \mathbb{N} \times \{s\}$. Thereby, $\mathcal{C}$ is the set of all concepts, $\mathcal{P}$ the set of all properties as well as $\mathcal{T}$ the set of all terms, and $\mathbb{N}$ the natural numbers.

Hierarchy representation

For query expansion and other query operations, we need to represent concept and category hierarchies in the index. Here, we use the Dewey Identifiers [Dew04] ap-

proach [OOP+04, HHMW07, AYCD06, BG06, XP05]. Figure 6.6 illustrates the Dewey approach. We add two new nodes to the concept schema, “concept” and category with the identifiers 1 and 2, respectively. All root concepts are direct children of node “concept” and all root categories are direct children node “category”. According to the Dewey numbering scheme, a node has an identifier of the form *prefix.suffix*. The prefix is the identifier of the parent node. The suffix of the identifier is the sibling number by numbering the siblings from left to right. For example, the concept “CulturalAsset” has the identifier 1.2 because the parent is the concept root (id = 1), and the ‘CulturalAsset” is the second child (“Person” is the first). All concepts start with 1 and all category identifiers with a 2 by using this scheme.

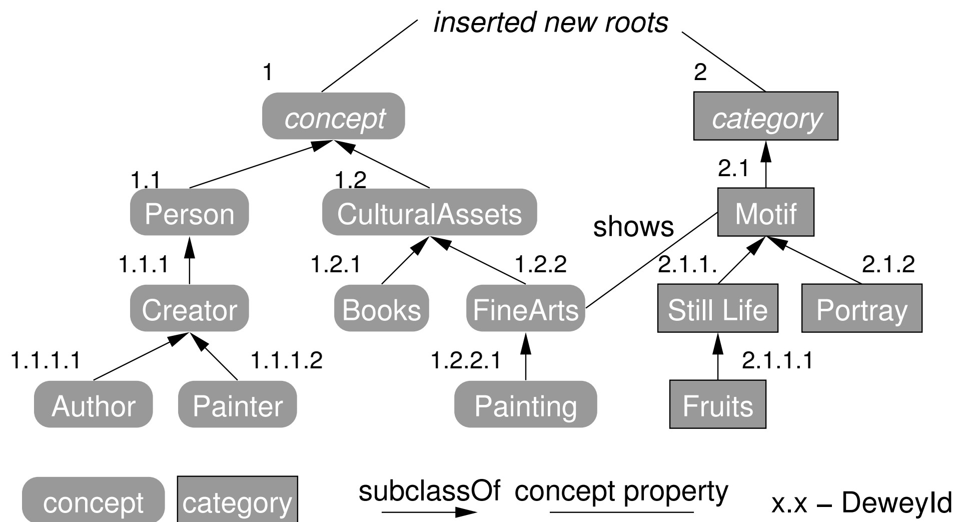


Figure 6.6.: Dewey numbering

Keyword index structure

The keyword index comprises term positions occurring in the data sources as well as in the global concept schema and mapping information. Thereby, the index follows the inverted list paradigm storing for each term a list of occurrences [ZMR98]. The basic structure of a keyword index entry is illustrated in Figure 6.7.

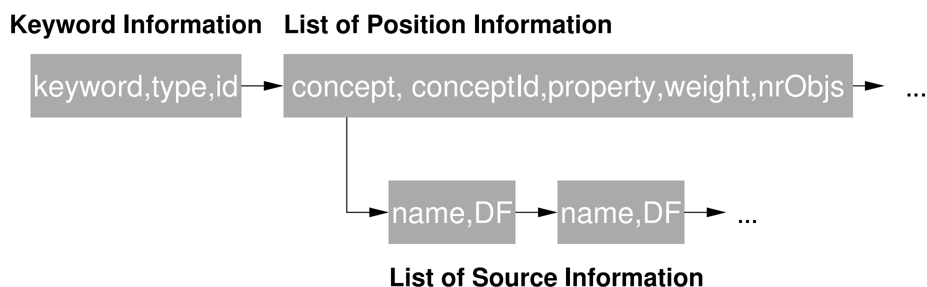


Figure 6.7.: Index entry structure

A keyword index entry consists of three parts: *(i)* the term information, *(ii)* the list of position information, and *(iii)* a list of source information for each position. The term information is a triple $(t, type, id)$ with $t \in \mathcal{T}$ a term, *type* a flag describing the role of the term, and *id* a Dewey identifier or *null*. We define five different possible types of a term according to its position: concept (c), category (v), data property (dp), concept property (cp), and data d. Table 6.1 summarizes the meanings of the types. The identifier *id* is the Dewey identifier of the corresponding category, if the term is of type category (v). Otherwise, the identifier is omitted. Each term information entry

Type	**Description**
concept c	occurrence on concept level; term in concept name or source description for a concept
data property dp	occurrence on concept level; term in a data property name or source description of a data property
concept property cp	occurrence on concept level; term in concept property name
data d	term occurs in a data source
category v	term occurs in a data source but represents a category on concept level or a local instantiation of it

Table 6.1.: Index term types

refers to a list of position information. The position information of a term consists of the *concept*, the corresponding concept identifier (*conceptId*), and the *property* as well as the *weight* of the term in this position as well as the number of objects (*nrObjs*) containing the term. The position information contains a list of source information describing the term occurrences in every source. The source information list is empty for concept and property keywords. The concept information is null for concept property keywords, because these keywords describe connections between concepts, i.e., relationships. The information *nrObjs* is only valid for category and data terms. The source information is a pair (s, df), the name of the source s and the document frequency $df(t, c, p, s)$ of the term in that source for the given position. It is obtained from the source content descriptions.

Figure 6.8 shows an exemplary index part. The example contains different terms like painter, holbein, fruits, etc. For example, the keyword "fruits" occurs in two roles, as category term or value term, respectively. Furthermore, the example illustrates that concept level terms ($type \in \{c, cp, dp\}$) do not have a source list, while other terms occur in different places and different sources. In Section 8.1, we describe the implementation of the keyword index as one relation in a relational database.

6.2.2. Index Lookup Methods

The method **indexLookup** provides access to the index. The input to the method is either a concept-based or a plain keyword query. The output is a list of index entry tuples. An index entry consists of the keyword information and one position

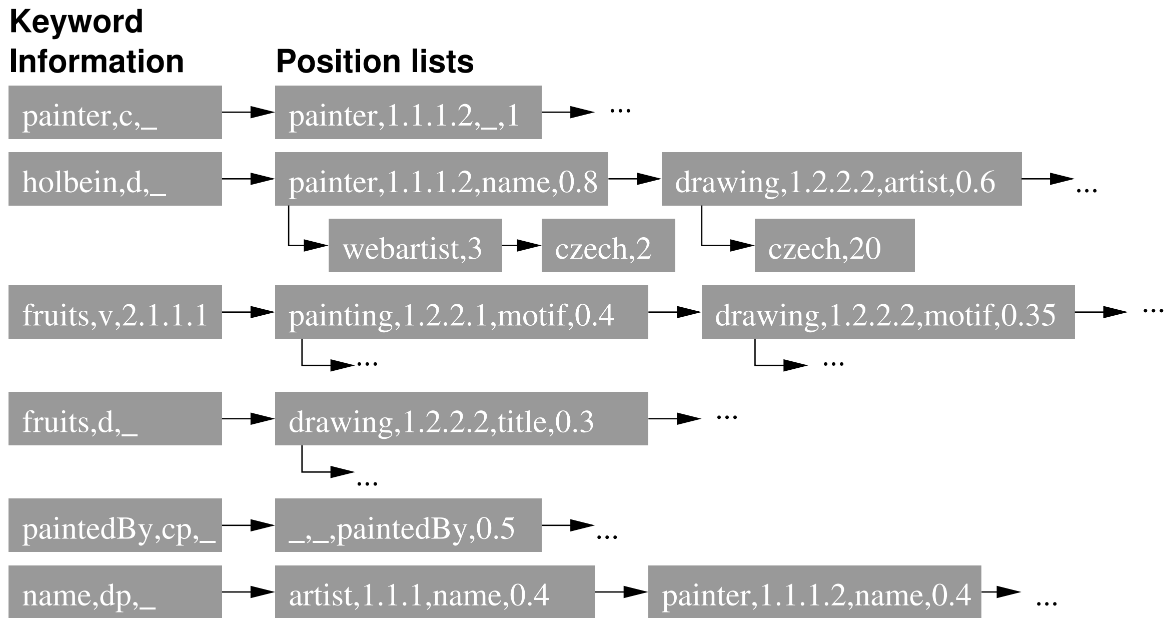


Figure 6.8.: Index example

information. Every position in a tuple refers to the same concept or the concept property. That means that we merge the position lists of the keywords. In a following step, every tuple is translated into a single concept query, and the score of this query is computed.

Figure 6.9 outlines the processing of a single query term. In this process, we distinguish two cases: plain and concept-based query terms.

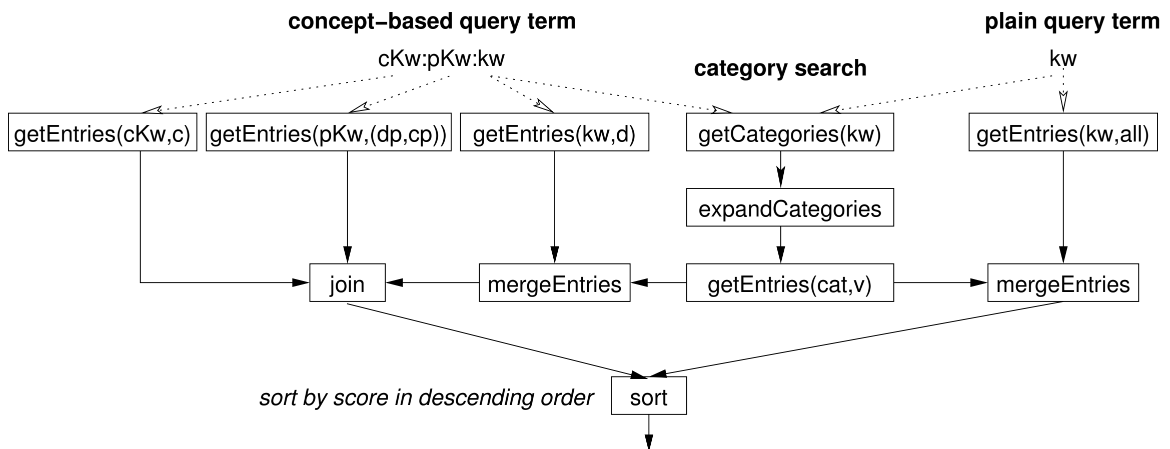


Figure 6.9.: Single query term lookup

Plain query term. For a plain query term, the system obtains all index lists that contain the keyword for all roles except the role category using the function **getEntries**. The keyword search system supports categories in the following way. At first, all categories are obtained for the keyword using function **getCategories**. After that,

the system constructs semantically related categories and their respective scores (see Section 5.4.3) using function **expandCategories**. For this, we get all ancestors of a selected category v, which are identified by the prefixes of the Dewey identifier of v. We extract all categories that have a common ancestor with v, i.e., they have common Dewey prefixes. We compute the score according the semantic distance and the equation (5.18). Thereby, we remove duplicates in the lists, keeping the better score. Finally, we retrieve all value entries for the category, i.e., index entries of the type v. In the following step, we merge all lists and sort the result by concept, property, and descending score.

Concept-based query term. A concept-based query term is a triple $t = (kw^c : kw^p : kw^v)$. For each non-null component, the system executes the **getEntries** function. For the concept-label kw^c, **getEntries** is executed with the condition c indicating only concept position must be returned, for the property keyword kw^p, entries for property (dp) and concept properties (cp) are returned. The value keyword is handled in two steps. On the one hand, the value keyword kw^v is used with option d to retrieve its position as a literal value. On the other hand, we use the category approach as described above to find categorical values. We merge the lists to one value index entry list. Now, we have entry lists for every non-null component. We join these lists to get entry tuples representing the index lookup result for concept-based query terms. The entries of one tuple must have the same concept as well as the property and the value entry have to refer to the same property.

Concept label expansion. In the case of concept expansion, we use the following idea. Let us assume the query term $drawing : title : flowers$ with the index entries for $L[drawing ::] = [(drawing, 1.2.2.2)]$, $L[: title :] = [(painting, 1.2.2.3, title)]$, and $L[:: flowers] = [(painting, 1.2.2.3, title)]$. Using query expansion, we combine the results. First, we combine property and value keywords. For example, we join $L[title]$ and $L[flowers]$ to $L[(: title : flowers)]$. Now, we join $L[drawing ::]$ and $L[(: title : flowers)]$ in the following way. For every entry tuple in $L[(: title : flowers)]$, we find the best match in $L[drawing ::]$ that has the following characteristics:

- the concepts have a common ancestor in the same concept hierarchy, and
- it has the highest score of all possible matches.

For example, the common ancestor is 1.2.2 that is the concept "fine arts". The corresponding score of the joined tuple is adapted according to the semantic distance between drawing and painting. The distance is computable using Dewey identifiers and the hierarchy height. The result of the example is $L[(drawing : title : flowers)] = [((drawing, 1.2.2.2), (painting, 1.2.2.3, title), (painting, 1.2.2.3, title))]$.

The result of this join is a list of index entry tuples that are valid for the concept-based query term. The list is sorted by concept, property, and descending score.

6.3. Query List Generation

A *query list* comprises single concept queries ordered by descending score. All queries in one list refer to the same concept (or concept property) and contain the same set of keywords. The *query list generator* component takes an index entry list for every keyword as input. It processes and combines the lists into single concept query lists for all combinations of concepts (concept properties) and keyword query subsets. For every query list, the query list generator adds representing concept nodes and edges into the schema graph of the virtual document. The query lists and the annotated concept schema graph are the result of query list generator and are the input of the following step, the query list network generation. The query list generator creates the result in three steps:

1. It creates index entry tuples lists for every concept/keyword set combination or concept property/keyword set combination, respectively.

2. It creates single concept queries and their score from index entry tuples.

3. It creates an annotated concept schema graph.

6.3.1. Index List Generation

The index list generation generates for every supported concept and every possible keyword query subset an index list. The input for this task is the concept schema graph and $|Q|$ index entry tuple lists $L[kw_1] \dots, L[kw_n]$. The output is a data structure $\mathcal{G}$ containing triples $t = (c, Q', l)$ with c a concept or edge in the concept graph, Q' a subset of the keyword query Q, and l a list of index entry tuples t. Every component $t[kw]$ in these tuples refers to one keyword in Q'. Algorithm 6.1 outlines the approach.

In the first step, we group every input list $L[kw_i]$ into different parts. Every group, denoted as $\mathcal{G}[c, \{kw_i\}]$, contains index entries assigned to the same concept (concept property)[1] c (lines 3-6). In the following, we combine these lists separately for every concept c (lines 8-20). For every concept that has more than one keyword, we compute all Cartesian products of the corresponding query lists. The Cartesian products describe all possible combinations of keyword interpretations for the concept. The computation is stepwise. First, we compute all lists for two keywords. Subsequently, the algorithm adds a list to the previous computed results, avoiding double computations (lines 11-18). At the end, the algorithm has computed all combinations and returns the data structure $\mathcal{G}$.

Example 6.1 *For the plain keyword query “holbein prague” we illustrate the index entry list generation in Figure 6.10. Initially, there are two index entry tuple lists, $L[holbein]$ and $L[prague]$. The lists are separately grouped by concept, and the results are stored in the data structure $\mathcal{G}$. For the concepts “drawing” and “painting”, we combine the lists $\mathcal{G}[drawing, \{holbein\}]$ and $\mathcal{G}[drawing, \{prague\}]$ to $\mathcal{G}[drawing, \{holbein, prague\}]$ as well a $\mathcal{G}[painting, \{holbein\}]$ and*

[1] The following discussion describes only query lists for concepts. Concept property lists have always one entry but the computation is equivalent.

Algorithm 6.1 Query list generation

Input: $L[kw_1], \ldots, L[kw_n]$ – index entry tuple lists with $n = |Q|$ the number of query terms
(C, E) – the concept graph
Output: $\mathcal{G}$ – all query lists

```
1: function QueryListGeneration(L[kw_1], ..., L[kw_n])
2:     G := ∅
3:     /* group every list by concept/property into G */
4:     for 1 ≤ i ≤ n do
5:         group L[kw_i] into G[, {kw_i}] /* one list for each concept and keyword */
6:     end for
7:     /* Combine all groups concept-wise */
8:     for c ∈ (C ∪ E) do
9:         if |G[c]| > 1 then /* more than one keyword */
10:            let Q_c be the supported keywords in G[c]
11:            for 1 ≤ i < |Q_c| do
12:                for all pairs (kw, Q''); kw ∈ Q, Q'' ⊂ (Q_c \ {kw}), |Q''| = i do
13:                    Q''' = {kw} ∪ Q''
14:                    if G[c, Q'''] = ∅ then
15:                        G[c, Q'''] = G[c, {kw}] × G[c, Q'']
16:                    end if
17:                end for
18:            end for
19:        end if
20:    end for
21: end function
```

$\mathcal{G}[painting, \{prague\}]$ to $\mathcal{G}[painting, \{holbein, prague\}]$, respectively. The union of the lists of the first and second step forms the final set of index entry lists in $\mathcal{G}$.

6.3.2. Single Concept Query Generation

We now describe the translation of an index entry tuple t to a single concept query q and the computation of the score value $score(q, Q)$. The input of the algorithm is the tuple t, the represented query subset $Q' \subseteq Q$, and the query Q. The output is a single concept query q of the form $\sigma_{cond}(\mathbf{ext}(c))$. The condition $cond$ is a conjunction of predicates. The form of a predicate is either $exists(p)$, $p \sim= kw$, or $p = k$ with p a property, kw a term, and k a category. The algorithm is applied to all index entry lists. Algorithm 6.2 sketches the approach.

An index entry tuple consists of $|t|$ entries. For every plain query term, there is one index entry in the tuple as seen in Figure 6.10. In the case of a concept-based query term ($concept : property : value$), the tuple has three entry components, one for each component. We continuously number every tuple component. A component is accessed by $t(i)$ with $1 \leq i \leq |t|$. The algorithm scans over all tuple components. It

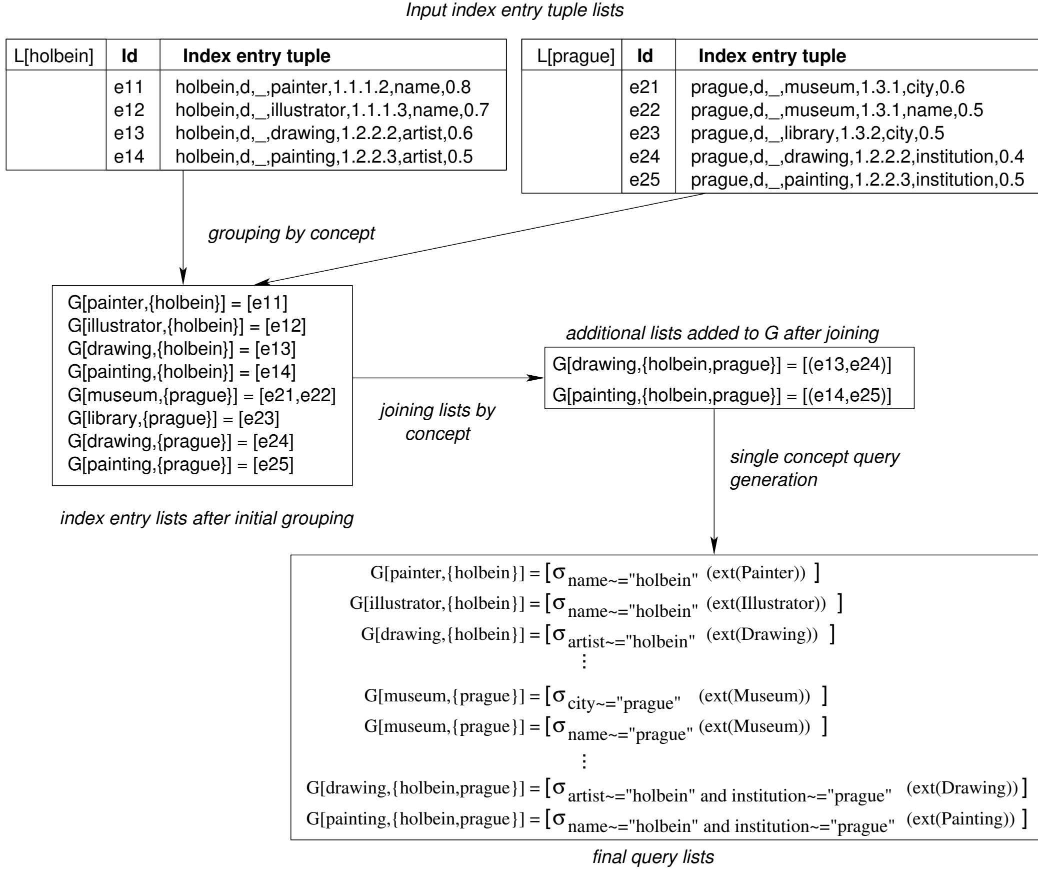


Figure 6.10.: Query list generation

creates the query condition and summarizes the schema and data score of the query. For every entry $t(i)$ for $1 \leq i \leq |t|$, the algorithm tests the type of the keyword. Algorithm 6.2 shows the actions for every type. The first three cases (c,dp,cp) (lines 5-14) do not change the *dataScore* because they deal with schema level keywords. The special case of a concept property (cp) sets the query to *empty*, which is a marker for a selected concept property. In the case of a data property (dp), the algorithm adds a predicate exists($t(i).property$) to the select condition, i.e., the results have to have a value for this property. If the keyword is either a category (v) or data (d) term (lines 15-23), the algorithm will add the corresponding predicate to the condition and update the data score. Finally, the algorithm uses the function **computeScore** to obtain the score of the generated query including the compactness of the query. Note, that the weight $t(i).weight$ of every entry already includes the query term weights.

Using the function **SingleConceptQueryGen** the system replaces all index entry lists in $\mathcal{G}$ by lists of pairs $(q, score(q, Q))$. The single concept query lists in $\mathcal{G}$ are the input of the concept graph annotation described in the following section.

Example 6.2 *We continue the example in Figure 6.10. Given the index entry tuple lists* $\mathcal{G}$, *the in tuples are translated into single concept queries. Assume the list* $\mathcal{G}[drawing, \{holbein, prague\}] = [(e_{13}, e_{24})]$ *as an example. The tuple* (e_{13}, e_{24}) *contains two data entries and relates to the concept drawing. The first entry is*

Algorithm 6.2 Single Concept Query Generation

```
    Input: t – index entry tuple
           Q – the complete keyword query
    Output: q – single concept query
           score(q,Q) – the score of q according to Q
 1: function SingleConceptQueryGen(t,Q)
 2:     cond := true, c := null
 3:     schemaScore := 0, dataScore := 0
 4:     for 1 ≤ i ≤ |t| do
 5:         if t(i).type = c then /* concept */
 6:             c := t(i).concept
 7:             schemaScore := schemaScore + t(i).weight
 8:         else if t(i).type = dp then /* data property */
 9:             schemaScore(t) := schemaScore + t(i).weight
10:             c := t(i).concept
11:             cond := cond ∧ exists(t(i).property)
12:         else if t(i).type = cp then /* concept property, only marker query */
13:             q := empty /* special query as marker */
14:             schemaScore(t) := schemaScore + t(i).weight
15:             dataScore(t) := 0
16:         else if t(i).type = d then /* data */
17:             c := t(i).concept
18:             cond := cond ∧ t(i).property ∼= t(i).keyword
19:             dataScore := dataScore + t(i).weight
20:         else if t(i).type = v then /* category */
21:             c := t(i).concept
22:             cond := cond ∧ t(i).property = getCategory(t(i).id)
23:             dataScore := dataScore + t(i).weight
24:         end if
25:     end for
26:     if q ≠ empty then
27:         q := σ_cond(ext(c))
28:         score(q,Q) := computeScore(schemaScore, dataScore, t)
29:     else/* concept property case */
30:         score(q,Q) := schemaScore
31:     end if
32:     return q,score(q,Q)
33: end function
```

translated to the predicate artist ∼= "holbein" and the second to the predicate institution ∼= "prague". Together, the tuple is translated into the single concept query

$$\sigma_{artist \sim= "holbein" \wedge institution \sim= "prague"}(\boldsymbol{ext}(Drawing)).$$

This query retrieves all drawings from sources that have an artist value containing "holbein" and an institution value containing "prague". We omitted the score values in the example.

6.3.3. Concept Schema Graph Annotation

The concept schema graph, i.e., the schema of the virtual document, contains all concepts and their connections. Every concept represents a set of global objects called extension. We now describe the *annotated concept schema graph* that adds *annotated* nodes and edges to the graph that represent query lists. In turn, a query list of a concept represents a subset of the concept extension.

A concept graph is denoted as $CG(D) = (C_{CG}, E_{CG})$ for a virtual document $D = (N_D, E_D)$ (see Section 5.2). The annotated concept schema graph extends this definition to $SG_A = (C_A, E_A, ann)$ with $ann : C_A \cup E_A \to \mathbb{P}Q$ assigns keywords of a given query Q to the nodes and edges of the graph. We denote the annotation of a node c as $c^{ann(c)}$. The original nodes of the input concept graph are annotated with the empty set, i.e., $c^{\emptyset}$. The schema graph and the query lists in $\mathcal{G}$ are the input of the concept schema graph annotation algorithm.

For every query list $\mathcal{G}[c, Q']$ with c a concept and $Q' \subseteq Q$ a keyword set, we create a new concept node $c^{Q'}$ in SG_A. The new node $c^{Q'}$ is a copy of the original concept $c^{\emptyset}$. That means that it is a sibling of $c^{\emptyset}$ in the concept hierarchy and all concept properties from and to $c^{\emptyset}$ are also concept properties from and to $c^{Q'}$. The new node $c^{Q'}$ represents the query list $\mathcal{G}[c, Q']$ and a subset of the concept c.

For every query list $\mathcal{G}[e, Q']$ of a concept edge $e \in E$, we create a copy $e^{Q'}$ of the edge $e^{\emptyset}$ and add it to the edge set E_A. The graph indicates where keywords have been found and allows the construction of query list networks that are concept networks with annotated nodes and edges.

Example 6.3 *We continue the example illustrated in Figure 6.10. Given the query lists in* $\mathcal{G}$*, the system generates an annotated concept graph (see Figure 6.11). Thereby, the concepts painter, illustrator, drawing, painting as well as museum and library are copied and annotated with the respective keyword sets. For example, the node* $drawing^{\{holbein,prague\}}$ *represents the query list* $\mathcal{G}[drawing, \{holbein, prague\}]$*. In this example, we omitted the duplicate concept properties "created by" and "lost by" for better readability and show only the connection for the top concepts.*

6.4. Query List Network Enumeration

In the following step, the enumeration of query list networks is a preparation step to create queries to materialize object networks. A query list network is a concept

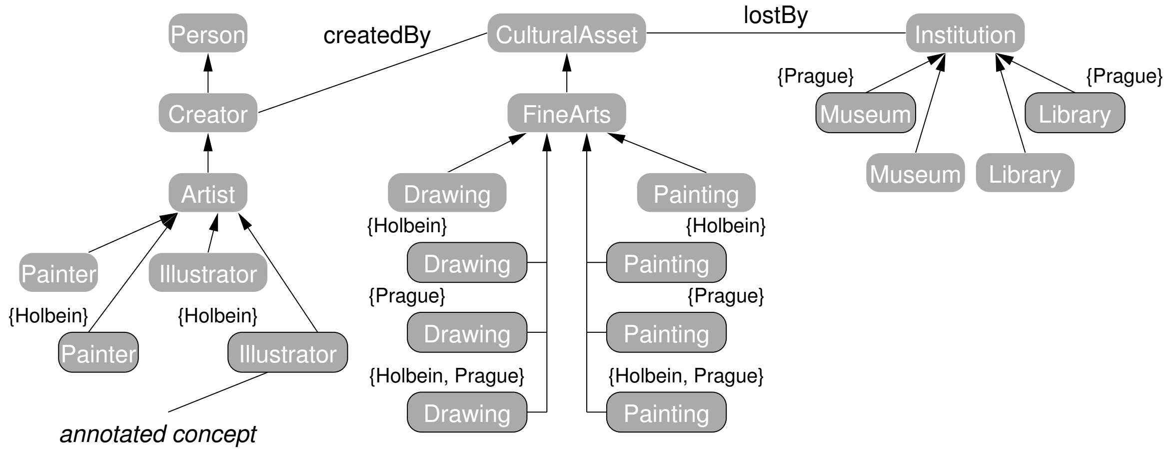


Figure 6.11.: Concept graph annotation

network that consists of query list concepts and edges as well as free concepts and edges.

Furthermore, we motivate the enumeration algorithm. We show the challenges and introduce the compact annotated concept schema graph. The compact schema graph condenses concept hierarchies in order to decrease the graph complexity. Based on the condensed graph, the enumeration algorithm is outlined.

6.4.1. Query List Network

In Section 5.2, we defined object networks and concept networks as results of keyword queries. A *query list network* is a concept network (see Section 5.2) over an annotated concept schema graph. The query list network comprises *query list concepts* and *free concepts*. Free concepts represent complete concept extensions and have an empty keyword set. In turn, query list concepts represent a certain subset of the keywords and single concept query list. Furthermore, edges can be either free (with empty keyword annotation) or specified (with non-empty keyword annotation). A query list network represents a set of materialization queries. A materialization query is constructed by combining single concept queries and the structure of the query list network. Given are an annotated schema graph $SG_A = (C_A, E_A, ann)$ and a query Q. A query list network $qln = (A_{qln}, E_{qln})$ can create valid materialization queries, if it has the following properties:

1. qln contains all query terms of Q, i.e.,

$$\left(\bigcup_{(a,c)\in A_{qln}} ann(c) \right) \cup \left(\bigcup_{e\in E_{qln}} ann(e) \right) = Q,$$

2. every query term is included exactly once, that means that $\forall t, t' \in (A_{qln} \cup E_{qln})$: $ann(t) \cap ann(t') = \emptyset$ with $t \neq t'$, [2]

3. all leaves in qln have to be specified. A leaf $t = (a, n) \in A_{qln}$ is specified, if
   - the annotation is non-empty $ann(t) \neq \emptyset$, or
   - there is an edge $e = (a, t) \in E_{qln}$ or $e = (t, a) \in E_{qln}$ with $ann(e) \neq \emptyset$.

4. two edges $e = ((a_1, c_1), (a_2, c_2))$ and $e' = ((a_3, c_1), (a_2, c_2))$ must not occur in E_{qln} if the edge (c_1, c_2) in E_A is an N:1 or 1:1 edge [HGP03].

The first requirement ensures the AND semantics of the keyword query. The second point minimizes the overlap between query list networks. For example, consider the query list network $drawing^{Prague,Holbein} - lostBy - institution^{Prague,Gallery}$ with overlapping keywords. The results of this set of queries are also contained in the query list networks of $drawing^{Holbein} - lostBy - institution^{Prague,Gallery}$ or $drawing^{Prague,Holbein} - lostBy - institution^{Gallery}$, respectively. That means that we map every keyword to exactly one concept or edge in a query list network. The third point ensures the creation of minimal materialization queries, with that minimal object networks.

Object networks have to comprise distinct objects. In order to ensure this requirement, we exploit the structural information about the edges [HGP03]. Assume an edge *createdBy* between the concept *painter* and *paintings* with the cardinalities 1:N. That means, every painter might have painted N paintings, but every painting is created by only one painter. Let us assume we created a query list network $(a_1, painter) - createdBy - (a_2, paintings) - createdBy - (a_3, painter)$. Then, this query list network would only create invalid object networks, because a painting is always connected to only one painter. Thus, point four allows discarding such kind of query list networks.

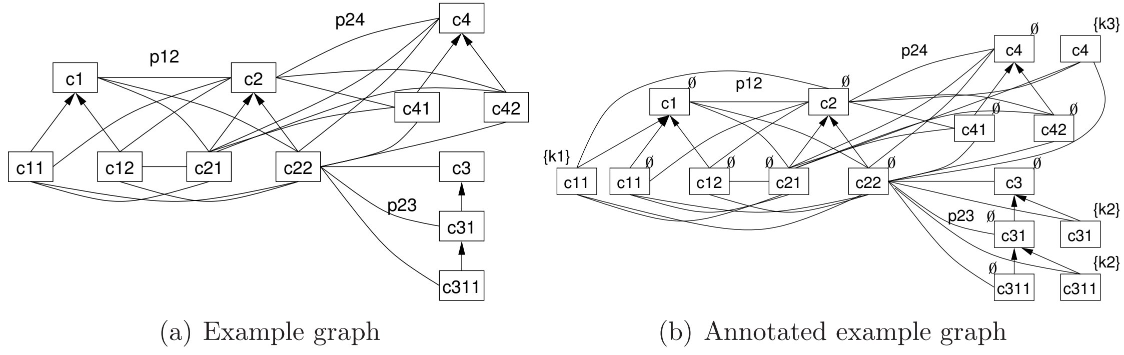
Nevertheless, a query list network still can create invalid queries. This is the case, if all single concept queries are schema queries. These invalid queries are filtered out during the execution phase.

6.4.2. Compact Concept Schema Graph

Given the definition of a query list network, we describe the efficient enumeration of them using an annotated schema graph consisting of query lists and free concepts. Because of many different concepts and inherited concept properties, the schema graph is extremely complex. For example, Figure 6.12(a) shows a graph with all connections. Annotation introduces new nodes and edges into the graph and increases the complexity further (see Figure 6.12(b)).

In order to limit the enumeration time, we have to reduce the complexity of the annotated schema graph. Therefore, we introduce the compact concept schema graph of an annotated schema graph. The *compact schema graph* is denoted as $CSG(SG_A) = (C_{csg}, E_{csg})$ where SG_A is the original, annotated graph. The compact schema graph

[2] We assume the annotation of an alias in the query list network equals the keyword annotation of the underlying node, i.e., $ann((a, c)) = ann(c)$.



(a) Example graph (b) Annotated example graph

Figure 6.12.: Example graph and corresponding annotated graph

contains *complex nodes* and *complex edges*. A *complex node* is a schema graph node representing the concept and its direct and indirect sub-concepts. All sub-concepts must have the same keyword annotation, too. Edges with the same label and keyword annotation are collapsed into one *complex edge*. The edges connect only complex nodes. That leads to the following problem. Consider the Figure 6.12(b) and the edge p_{23} between concept c_{22} and c_3. This edge is only valid for the concept c_{22} and its sub-concepts. Hence, we add a rule to the complex node $p_{23} \rightarrow c_{22}$. This rule says, if the incoming or outgoing edge is the concept property p_{23}, than only concept c_{22} and its sub-concept are allowed in the resulting materialization queries. These rules are stored for every complex node n in the set $er(n)$. For a complex node $n \in C_{csg}$, we add for all incoming and outgoing edges of the concept and its sub-concepts to the rule set to $er(n)$. For example, the complex node c_2 in Figure 6.13 contains the edge rule $p_{23} \rightarrow c_{22}$ in the rule set $er(c_2)$.

Given an annotated concept schema graph $SG_A = (C_A, E_A)$, the compact schema graph $CSG(SG_A) = (C_{csg}, E_{csg})$ is created in the following steps:

1. Add all unconnected nodes of C_A to C_{csg} and remove them from C_A.
2. Add the most general nodes from C_A to C_{csg} as complex nodes. A most general node is a node that does not have a super-concept with the same keyword set. This can be easily decided by using Dewey identifiers.
3. All other nodes are represented by the corresponding complex node. If a query list node is represented by a complex node, its corresponding single concept query list is merged into the complex node.
4. Let $c_1, c_2 \in C_A$ be concepts in the schema graph and $c'_1, c'_2 \in C_{csg}$ the corresponding complex nodes. Then, add the edge $e = (name, c'_1, c'_2)$ to E_{csg} for the edge $(name, c_1, c_2) \in E_A$. Duplicate edges are removed. If c_1 is a sub-concept of c'_1 and there is no edge $(name, c''_1, c_2)$ with c''_1 a super-concept of c_1, then add a rule $name \rightarrow c_1$ to $er(c'_1)$.

Figure 6.13 illustrates the resulting compact graph for the example graph in Figure 6.12(b). The most general nodes are $c_1^{\emptyset}$, $c_2^{\emptyset}$, $c_3^{\emptyset}$, and $c_4^{\emptyset}$. Additionally, we have to add the nodes $c_{11}^{\{k_1\}}$, $c_4^{\{k_3\}}$, and $c_{31}^{\{k_2\}}$ as most general annotated nodes. These nodes

represent all of their sub-concepts with the same annotation. For example, the query lists of $c_{31}^{\{k_2\}}$ and $c_{311}^{\{k_2\}}$ are merged into one list represented by $c_{31}^{\{k_2\}}$. The complex node c_2 contains the rule $p_{23} \rightarrow c_{22}$ induced by the edge p_{23}.

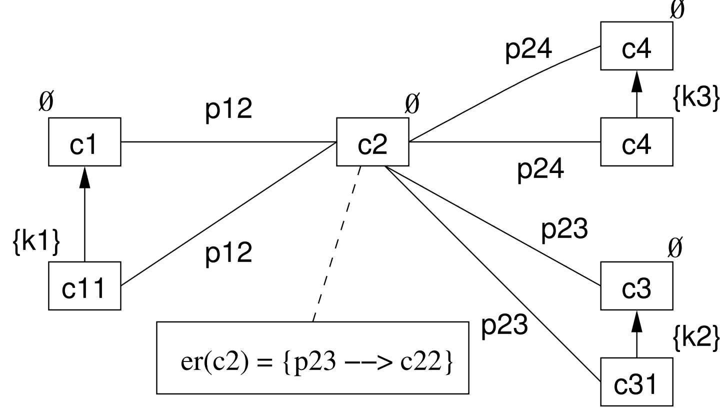


Figure 6.13.: Annotated compact example graph

6.4.3. Enumeration of Query List Networks

The goal of the next step is the enumeration of query list networks using a compact, annotated concept graph $CSG(SG_A)$. The graph contains free nodes and query list nodes as well as free edges and specified edges. We use a breadth-first search (BFS) algorithm following the related schema graph-based approaches [HP02, SLDG07, LYMC06]. Furthermore, we use several rules to create valid query list networks that eventually generate valid object networks. Algorithm 6.3 outlines the approach.

The inputs of the algorithms are the compact schema graph $CSG(SG_A) = (C_{csg}, E_{csg})$, the maximum size of query list networks $size_{max}$, and the keyword query Q. The algorithm starts with the initialization of the *queue*. It selects all nodes that contain the first keyword of the query $kw_1 \in Q$ (line 5). For each of these nodes, it creates a query list network with one node $("c_1", n)$ with $"c_1"$ the alias. If the network is already a result (line 7), then we will add the network to the output QLN. Otherwise, we add the network to the queue. Subsequently, the extension steps are executed until no networks can be extended anymore, i.e., the queue is empty (lines 14 to 28). The extension of a network using the breadth first approach works as follows. We retrieve the first query list network $qln = (A_{qln}, E_{qln})$ from the queue and remove it. For every node in A_{qln}, we look for all adjacent nodes n' to node n in the compact concept graph. Using the function **extend** we extend the qln with the edge $(n, n') \in E_{csg}$ and the new node $n' \in N_{csg}$. The function **extend** adds a new node $("c_x", n')$ with $x = size(qln) + 1$ and a new edge between (a, n) and $("c_x", n')$ to a copy of qln. Finally, it returns the new network $newQln$. In the next step, we will add $newQln$ to the result set, if it is a valid result, otherwise it is added to the queue, if it can form a valid result in future steps.

We now explain the help functions **appendTo**, **isResult**, and **isValid** . The function **appendTo** adds a network to the enumeration queue or to the result. The function

Algorithm 6.3 Query List Enumeration

Input: $CSG(SG_A) = (C_{csg}, E_{csg})$ – compact graph
$size_{max}$ – maximal size of the query list networks
Q – keyword query
Output: QLN – list of query list networks

1: **function** AND_QLN_ENUMERATE($CSG(SG_A)$,$size_{max}$,Q)
2: /* *initialize* */
3: $queue := [\,]$ /* *empty queue for breadth first search* */
4: $QLN := \emptyset$
5: **for all** $n \in C_{csg}$ with $kw_1 \in ann(n)$ **do**
6: create qln with one node $("c_1", n)$
7: **if isResult**(qln, Q) **then**
8: **appendTo**(QLN, qln) /* *qln is an output* */
9: **else**
10: **appendTo**$(queue, qln)$ /* *add to queue to extend* */
11: **end if**
12: **end for**
13: /* *Enumerate* */
14: **while** $queue.isNotEmpty$ **do**
15: $qln := queue.poll$ /* *remove first network from queue* */
16: **for all** $(a, n) \in N_{qln}$ **do** /* *For all nodes in* $qln = (N_{qln}, E_{qln})$ */
17: **for all** adjacent $n' \in C_{csg}$ to n **do**
18: $newQln :=$ **extend**$((a, n), (n, n'), n')$ /* *Extend qln* */
19: **if isValid**$(newQln, qln)$ **then**
20: **if isResult**(qln, Q) **then**
21: **appendTo**$(QLN, newQln)$ /* *is an output* */
22: **else if** $size(newQln) < size_{max}$ **then**
23: **appendTo**$(queue, newQln)$ /* *add to queue to extend* */
24: **end if**
25: **end if**
26: **end for**
27: **end for**
28: **end while**
29: return QLN
30: **end function**

ensures a duplicate-free queue or result. For this, we use an index in which we hold a canonical form of the query list networks. If the index does not contain the canonical string, the function will add the network to the queue or to the output and the string to the index. Otherwise, the network is ignored. Possible optimization are proposed in [MYP09, QYC11, Luo09], which improves performance further. We will discuss the optimization in the following section. The second function **isResult** will return true for a query list network qln, if the network contains all keywords of the query, i.e., $keywords(qln) = Q$ and the network contains at least one annotated concept node. The last function **isValid** checks if a query list network is valid according to the rules in Section 6.4.1. That means that there are no overlapping keyword sets, no violation of the 1:n rule, no violation of the free leaves rules, and the network does not violate the rules in the compact schema nodes. We explain the latter two points in the following.

Assume a query list network qln has k free leaves. The network is valid if the algorithm can extend it in that way that only specified leaves exist. That is only the case if $k \leq \min(maxSize - size(qln), |Q| - |keywords(qln)|)$. The condition states that the network must be expandable by at least k nodes. This is the case, when the maximum size is not violated if we add k nodes and there must not be overlapping keyword sets. Hence, at least k keywords of Q are not in the set.

The last point investigates compact nodes. A compact node does not support all combinations of edges. Therefore, a node has a set of rules $edge \rightarrow concept$ describing the most general concept that supports the edge. Now assume a node (a, n) in a query list network and let $edges((a, n))$ be the incoming and outgoing edges from this node. We use all rules for n and replace the edges by their mapped concepts, i.e., $concepts((a, n))$. The node supports the set of edges if for all pairs (c, c') with $c, c' \in concepts((a, n))$ is true that c and c' are in an ancestor-descendant relationship or are equal.

6.5. Discussion

The approach follows the schema-graph-based keyword processing in structured databases [ACD02, HP02, HGP03, SLDG07]. However, we use query lists instead of tuple lists that represent the selected objects. The data graph approach is not suitable because it requires the complete materialization of all global objects. Instead, the proposed approach uses only keyword statistics that can be obtained using extended protocols like STARTS [GCGMP97] and its extensions [GIG01, IBG02], or query based sampling approaches [CC01]. The statistics contain only information of single keywords but not information about the connection of terms. For example, Yu et al. [YLST07] describe connection statistics to be used in the selection of relational databases for a keyword query. However, we treat sources as a virtually integrated database. Thus, we try to select the global queries instead of directly selecting local relational databases.

The keyword index models concept and category hierarchies with the help of Dewey identifiers. Different works in XML [OOP+04, HHMW07] and XML keyword processing [AYCD06, BG06, XP05] successfully use Dewey identifiers to model the ancestor-

descendant relationship in XML hierarchies. We use the numbering to create complex nodes and for keyword processing with query expansions.

The keyword processing creates an interpretation of a keyword. For that, we support schema and data terms as well as faceted or labeled keywords. Many keyword search systems in relational databases support only data terms [HP02, ACD02, SLDG07]. Yu et al. [YLST07] also support schema terms during query processing. Other related systems are form search systems [CBC+09, DZZN09]. Here, pre-computed forms or query templates are associated with keyword interpretation. Tata et al. [TL08] also create queries from keywords, but go a step further. The authors create a subset of SQL queries from a keyword list and allow also aggregations. A similar idea is the form search of Chu et al. [CBC+09].

In the second step, our approach enumerates query list networks. Query list networks are equivalent to candidate networks [HP02, ACD02, SLDG07]. We use an online breadth first approach and do not precompute query templates [DZZN09] or forms [CBC+09]. In order to reduce the enumeration complexity, we condense the annotated schema graph. A similar approach is used in the KITE system [SLDG07]. However, we compact a concept graph by using hierarchies of concepts as well as their inter-concept edges, while the KITE system works with heterogeneous relational databases and compacts foreign key relationships. Furthermore, the compact networks allow to reuse the mediator query optimization of Yacob [SGS05] (see Section 3.3). Markowetz et al. [MYP09] and Qin et al. [QYC11] focus on keyword search on large data streams. The problem is the efficient generation of all possible candidate networks, i.e., all keyword combination can occur in every relation. Therefore, one needs more efficient candidate network enumeration algorithms. Markowetz et al. proposed an algorithm that sorts all relations and adds only relations that have larger identifier to existing intermediate candidate networks. Qin et al. improve this approach by using a template-based approach. A template represents a set of candidate networks. Thus, first all templates are generated, and second, the candidate networks are extracted. This approach resembles our compact concept graph model because compact query list networks contain all query list networks of all sub-concept combinations.

6.6. Summary

In this chapter, we described the schema graph-based processing of concept-based keyword queries. First, we outlined the overall keyword search process and described all steps and components. Our approach follows the schema graph-based approaches for relational and XML databases. Second, we presented the structure of the keyword index. In particular, we model the concept-hierarchies using Dewey identifiers, which allows the efficient testing of subclassOf relationships. The result of the keyword processor is a list of index entry tuples. In the following, we showed how index entries are translated into single concept queries. A single concept query allows retrieving objects from one concept extension. As keywords are spread over different objects, we have to generate join materialization queries. For this, we annotated the global concept schema, i.e., inserted concepts and edges that represent lists of single concept queries. From this, we used the common breadth-first algorithm, to create concept

networks that contain all keywords. Together with the query lists, these are query list networks. Because of the complexity of the concept schema graph, we presented an algorithm to compact the annotated schema by collapsing concept hierarchies into complex nodes and merge the corresponding query lists. The overall result of concept query generation is a set of query list networks. A query list network is a compact representation of a list of materialization queries. We expect that index access and query list network enumeration are efficient by using the proposed approaches. We report of the validation experiment results in Chapter 8. In particular, Section 8.3.2 contains experimental results of index access and query list network enumeration. In the following Chapter 7, we present different approaches how to process and execute query list networks to obtain valid object networks as results.

7. Concept Query Processing

In Chapter 6, we presented the efficient generation of materialization queries. The result of the approach is a set of query list networks. A query list network represents a lists of materialization queries. In this chapter, we focus on the efficient execution of query list networks. The execution of materialization queries is the most expensive step during keyword query processing because source queries have to be sent via network to local sources, be executed, and results must be retrieved and integrated. In this chapter, we present approaches to minimize the number of executed source queries and transferred objects (see Figure 7.1). This chapter focuses on join optimization, efficient execution of query list networks as well as techniques to exploit the structure of generated structural queries to avoid unnecessary re-computations.

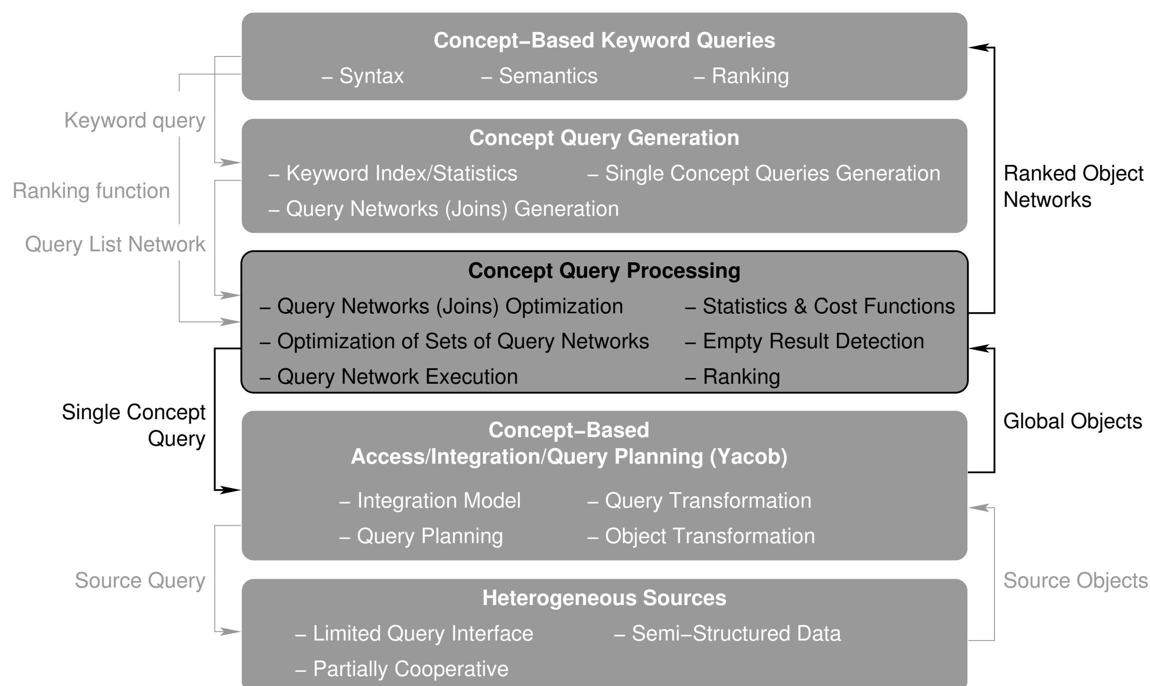


Figure 7.1.: System overview: Concept query processing

In Section 7.1, we discuss the query processing model. In order to optimize queries, we provide a cost model and its usage in a dynamic programming algorithm. Furthermore, we present the adaptation of existing keyword processing algorithms to concept-based keyword processing using materialization queries.

One problem of keyword query processing is the re-computation of queries. Section 7.2 describes how to avoid the re-computation of empty results. Section 7.3 discusses a semantic cache to materialize intermediate results for reuse of previously

computed results. Both approaches have the potential of reducing the cost, but are applied only to individual queries. Therefore, we present techniques to optimize a set of similar materialization queries in order to minimize the source accesses and queries in Section 7.4. For long query lists, we propose two optimizations: query merging and query splitting in Section 7.5. The chapter concludes with a discussion of related work in Section 7.6 and a summary in Section 7.7.

7.1. Preliminaries

This section is divided into three parts. First, we describe the join query processing. The processing uses the Yacob system to execute single concept queries. The results are taken over by the join processor that uses a semi-join approach. Queries are optimized by using a simple cost model and a dynamic programming algorithm. Second, we adapt two basic algorithms to process query list networks, i.e., lists of materialization queries created by the query transformation step. The first algorithm is the generation of all queries and sorting by score in descending order as proposed by Zhou et al. [ZZDN08]. The second algorithm is the step-by-step execution of query list networks defined by Hristidis et al. [HGP03]. Third, we motivate the following section by showing the degree of query overlapping and the problem of query re-computation.

7.1.1. Query Processing

In Section 5.3.3, we introduced materialization queries to obtain valid object networks as results of keyword queries. We also described how to translate them into concept-based join queries using the Yacob system. In the previous Chapter 6, we described approaches to generate such queries based on single concept queries. Now, we decompose a concept-based query into single concept queries and join conditions specified by concept properties. Thus, a materialization query is a query network

$$qn = (Q, E)$$

with Q a set of queries and $E \subset Q \times Q \times \mathcal{P}$ are edges between single concept queries representing concept properties. Every single concept query $q \in Q$ is of the form (see Section 3.2)

$$\sigma_{cond}(\biguplus_{c \in C_q} \mathbf{ext}(c))$$

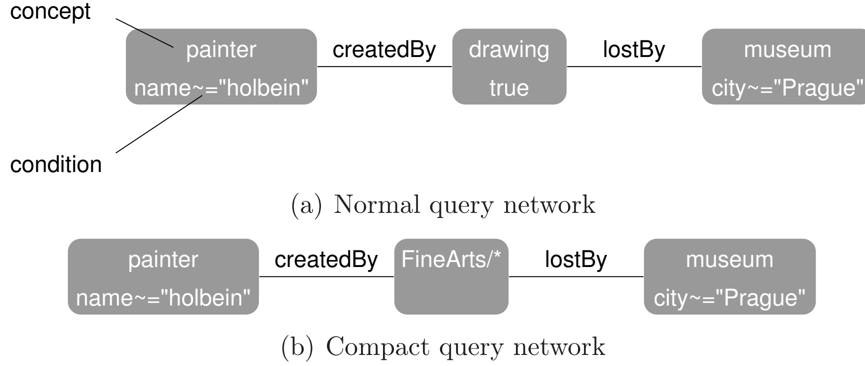
with

$$C_q = \begin{cases} \phi^{+}_{\mathbf{is_a}}(c) & \text{if sub-concepts are used} \\ \{c\} & \text{otherwise.} \end{cases}$$

That means that the query uses either the shallow concept extension or the union of the concept and all sub-concept extensions. If the condition *cond* is *true* or contains only concept level predicates like exists(p), we will denote the single concept query as *free*. Otherwise, we will say the query is *specified*. A specified query contains at least a predicate of the form $p \sim= kw$ or $p = v$ with kw a keyword and v a category. The

edges describe the connections between the query results. Using the join mappings the edges represent join conditions making a query network a conjunctive SPJ[1] query.

We show two query networks in Figure 7.2. Figure 7.2(a) illustrates a query network created from a non-compact query list network. In contrast, the query network in Figure 7.2(b) uses a compact node ($FineArts/*$). Every rounded box represents a single concept query. The upper text describes the set of concepts and the lower text is the condition. The middle nodes in both figures are free queries, the others are specified. The edges are mapped to the conditions $name = artist$ for "`createdBy`"



(a) Normal query network

(b) Compact query network

Figure 7.2.: Exemplary query networks

and $institution = name$ for "`lostBy`". Using this information, the query network of Figure 7.2(a) represents the join query

$$(\sigma_{name \sim = 'Holbein'}(\mathbf{ext}(painter))) \bowtie_{name=artist} \mathbf{ext}(drawing) \\ \bowtie_{institution=name} (\sigma_{city \sim = 'prague'}(\mathbf{ext}(museum)))\,.$$

The second network (Figure 7.2(b)) translates to the join expression

$$(\sigma_{name \sim = 'Holbein'}(\mathbf{ext}(painter))) \bowtie_{name=artist} \left(\biguplus_{c \in C_q} \mathbf{ext}(c) \right) \\ \bowtie_{institution=name} (\sigma_{city \sim = 'prague'}(\mathbf{ext}(museum)))$$

with $C_q = \Phi^{+}_{\mathbf{is_a}}(FineArts)$, i.e., all direct and indirect sub-concepts of the concept Fine Arts.

The result of query network execution is a set of object tuples. The tuple components are denoted by the alias of the query. The component values are the corresponding objects.

Semi-join processing

To compute the result of a materialization query, we propose a two-phase approach. In the first phase, the join processor materializes all objects for every single concept

[1]Select-Project-Join query

query that are necessary for the result computation. In the second phase, we join these objects to the result using common join algorithms on the global level.

In the materialization phase, we distinguish between free and specified single concept queries. Specified queries are sent to the YACOB mediator and are executed. In contrast, free queries cannot be directly processed. The solution is a semi-join. We use the result of a specified query and compute the semi-join with the free query. For example, the query network in Figure 7.2(a) can be translated into the query steps $p' = \sigma_{name \sim =' Holbein'}(\mathbf{ext}(painter))$, $m' = \sigma_{city \sim =' prague'}(\mathbf{ext}(museum))$, and $d' = \sigma_{name \sim =' Holbein'}(\mathbf{ext}(painter)) \rtimes_{name=artist} \mathbf{ext}(drawing)$. In the second step, we join p', d', and m'.

The approach allows the parallel materialization of different filter queries and the usage of efficient joins on the global level. However, an early join of materialized sub-plans might reduce the costs because of early stopping if the system encounters empty results.

Plan operators and statistics

The system translates materialization queries into a plan of operators that represent the underlying physical operations. An optimizer creates a plan that minimizes the query execution costs. As we consider sources with restricted query capabilities, we assume that we have to provide a selection condition to a source to obtain results. That means that the objects of the free queries cannot be retrieved directly but depend on the input of the specified join partners. We use a bind-join approach [HKWY97][2] to implement the semi-join. In the following, we explain the plan operators and the costs of the underlying query processing.

A *plan operator* represents an underlying physical operation and processing algorithm. The plan operators follow the idea of the Garlic operators [HKWY97].The following properties are common to all plan operators types(Figure 7.3). A plan operator has a list of children operators, i.e., 0 to n inputs. Every plan operator has a list of parents. Every plan operator p has an output schema, i.e., the structure of the output object tuples ($schema(p)$). A plan operator has an $alias(p)$. An alias represents a single concept query. An alias refers to the corresponding query node in the query network and to the alias in the corresponding concept network. Finally, a plan operator p has three cost values: the estimated number of source queries ($estSrcQ(p)$), the estimated number of transferred objects ($estSrcObj(p)$), and the estimated output size of the operator ($estSize(p)$). The values refer only to the cost caused by the operator p.

To compute the estimated costs, we use the statistics obtained from the keyword index. We restrict the statistics to allow the fast inclusion of new sources. Table 7.1 summarizes and describes the used statistics values.

We outline every plan operator in the following. We distinguish two kinds of operators, source accessing operators and global operators. Table 7.2 summarizes the used operators and the cost functions. First, we discuss the three source accessing plan operators.

[2] Also known as dependent join [GW00]

Figure 7.3.: Plan operator

Value	Description
$DF(t, c, p, s, deep)$	number of objects of concept c in source s containing term t in property p (if $deep = true$ including sub-concepts)
$DF(v, c, p, s, deep)$	number of objects of concept c in source s containing category v in property p (if $deep = true$ including sub-concepts)
$size(c, s, deep)$	number of objects of concept c in source s (if $deep = true$ including sub-concepts)
$dom(c, p, s, deep)$	number of distinct values in p of objects of concept c in source s (if $deep = true$ including sub-concepts)
$\lvert \mathbf{ext}(c) \rvert$	$\sum_{s \in S_c} size(c, s, false)$

Table 7.1.: Statistics values

Base concept plan operator. The *ConceptPOP* plan operator indicates an access to local sources through the extension of a concept or the extension of a set of concepts. The parameters of the operator are the concept to be accessed and a flag, if the sub-concepts should be used. The output schema consists of all data properties of the represented concepts. The operator alone does not induce source queries and transferred objects, but is used by filter and bind join operators. The estimated size is the size of the extension of the concept

$$estSize(p) = \lvert \mathbf{ext}(c) \rvert \tag{7.1}$$

or of the deep extension

$$estSize(p) = \sum_{c' \in \Phi^{+}_{\underline{\mathbf{is_a}}}(c)} \lvert \mathbf{ext}(c') \rvert. \tag{7.2}$$

Filter Operator. The *FilterPOP* represents the selection of the output of its only child operator. The FilterPOP always has one child that is a ConceptPOP. The Filter-POP inherits the output schema as well as the alias of the ConceptPOP. Additionally, the FilterPOP has a non-empty condition *cond*. Combined with the ConceptPOP

Name	**Type**	**OutputSchema**	**Estimated Size**	**Est. Src. Queries**	**Est. Src. Objects**
$ConceptPOP(c, inclSub)$	source	$props(c)$	Eq. (7.1) or Eq. (7.2)	0	0
$FilterPOP(Cond)$	source	$schema(firstChild)$	Eq. (7.3)	$\|sources(C)\|$	Eq. (7.3)
BindSemiJoinPOP(edge)	source	$schema(secondChild)$	Eq. (7.5)	Eq. (7.4)	Eq. (7.5)
MaterializePOP	global	$schema(firstChild)$	$estSize(leftPOP)$	0	0
CachePOP(CE)	global	$schema(firstChild)$	Eq. (7.7)	0	0
MultiWayJoinPOP	global	$\bigcup_{p \in children} schema(p)$	global join estimation	0	0

Table 7.2.: Plan operators

child, the FilterPOP represents a single concept query. If the condition *cond* does not contain a specified condition, the represented query will not be specified and cannot be executed directly. The following cost estimation holds for specified queries. The number of source queries induced by the operator is the number of sources that provide data to the concept. The number of source objects equals the size of the estimated output of the operator. We estimate the number of source objects using the document frequency DF of the query terms. For example, assume a condition $cond := artist \sim= "Holbein" \wedge title \sim= "Scence"$. Then for every source s and concept c, we estimate the number of objects containing *holbein* and *scene* as

$$\frac{DF(holbein, c, artist, s) \cdot DF(scene, c, title, s)}{size(c, s)}$$

with $size(c, s)$ the number of objects in the extension of c for the source s and DF is the document frequency of a term. The formula is based on the assumption keywords are independent. Summarizing all sources S_c and generalizing the condition to k predicates, we obtain the estimated number of objects as in Equation (7.3):

$$estSrcObj(p) = \sum_{s \in S_c} \left(\left(\prod_{(c/p_j \sim= kw_j) \in cond} \frac{DF(kw_j, c, p_j, s, deep)}{size(c, s, deep)} \right) \cdot size(c, s, deep) \right) \tag{7.3}$$

with $1 \leq j \leq k$. The estimated size of the output equals to the estimated number of retrieved objects. The same formula is also used for category selections. The parameter *deep* is true, if the sub-concepts are included, too. Otherwise, the parameter is false, and we use a shallow extension.

Bind Semi-Join Operator. In order to retrieve the output of free queries, we use the bind join operation represented by the *BindSemiJoinPOP* plan operator. A BindSemiJoinPOP has two inputs. The left input is a specified input, i.e., a specified FilterPOP, another BindSemiJoinPOP, or a CachePOP, which retrieves objects from the cache. The BindSemiJoinPOP has one parameter that is the join condition obtained from the concept property. The output schema of the plan operator is inherited from the right child operator. The BindSemiJoinPOP represents the value semi-join between the left filtered input and the extension represented by the right concept plan operator or free filter operator. It is implemented as follows. Let O_{left} be the result of the left input. The bind semi-join computes the output as described in Algorithm 7.1. For each object tuple of the left input, we create a selection query against the right input. The condition is created by the right hand side of the join condition and the respective value of the current object. The output is the union of all query values.

The estimated number of source queries depends on the estimated output size of the left operator and the number of sources that provide objects for the right input. In detail, we estimate that we create $estSize(leftChild)$ selection conditions. Every

Algorithm 7.1 Concept bind-semi-join

Input: O_{left} – left result, set of object tuples
$cond := (c_{left}/p_{left} = c_{right}/p_{right})$ – value join condition
$q_{right} = \sigma_{cond_r} \biguplus_{c \in C_{right}}(\mathbf{ext}(c))$ – right query
Output: *Output* – set of objects of the right concepts that can join O_{left}

1: **function** BindJoin(O_{left},*cond*,*q*)
2: $Output := \emptyset, T := \emptyset$
3: $V = \pi_{c_{left}/p_{left}}(O_{left})$ /* *Projection and duplicate removal* */
4: **for all** $v \in V$ **do**
5: $cond_t = p_{right} = v \wedge cond_r$
6: $Output := Output \cup \sigma_{cond_t} \left(\biguplus_{c \in C_{right}} \mathbf{ext}(c)\right)$
7: **end for**
8: return *Output*
9: **end function**

selection is sent to every concept of the right child and to every source. That means that we estimate the number queries caused by a bind join semi-join as

$$estSrcQ(leftChild) \cdot \left(\sum_{c \in C_{right}} |S_c|\right) \tag{7.4}$$

with C_{right} the concepts of the right child and S_c the number of sources mapped to $c \in C_{right}$. This estimation is conservative and most possibly an overestimation because of duplication elimination by the projection in the bind-join algorithm and optimizations of the mediator (see Section 3.3).

The number of retrieved objects is the sum of the query sizes. We estimate this number using the following heuristics. Assume an edge e between a concept c_1 and c_2. The join mapping of the edge leads to the condition $c_1/p_{11} = c_2/p_{21}$. The edge has the cardinalities $e.srcCard = 1$ and $e.tgtCard = N$. In our definition, this means that all values in c_1/p_{21} are also in c_1/p_{11} but not vice versa. For example, all artists values in *paintings/artist* are also in *artist/name*. This assumption does not hold in the integrated case, but we use this assumption for estimation. Because of this, we can estimate the domain size $dom(c_1, p_{11}) = |\,\mathbf{ext}(c_1)|$. Because of the inclusion assumption, it holds $dom(c_1, p_{11}) = dom(c_2, p_{21})$. In consequence, we compute the domain size of c_1/p_{11} and c_2/p_{21} as the extension size of the "1" side of the edge. We denote this as $dom(c_{dom}, p_{dom})$. Using the join formula of Ullman [Ull90] on page 650, we estimate the output size as

$$estSize(p) = estSize(leftChild) \cdot \sum_{c \in C_{right}} \sum_{s \in S_c} \frac{size(c,s)}{dom(c_{dom}, p_{dom})}. \tag{7.5}$$

Furthermore, we assume that the output size equals to the number of transferred objects, i.e.,

$$estSrcObj(p) = estSize(p). \tag{7.6}$$

We evaluate these basic estimators in Chapter 8.

Materialization Operator. The first global plan operator is the materialization operator *MaterializePOP*. This plan operator has one child plan operator as input that is either a filter or bind semi-join operator. The MaterializePOP inherits the output schema of its child plan operator. The materialize operator represents the materialization of the objects created by the sub-plan rooted by the child. As it does not induce new source queries and changes the output, it inherits all cost values of the child node.

The data is either temporarily materialized for the query, or it is added to the cache.

Cache operator. The cache plan operator *CachePOP* is the counterpart of the MaterializePOP. A MaterializePOP materializes the result of source queries on the global level. A CachePOP retrieves data from the cache. It combines several cache entries E and a filter $filter_{cond}$ to the final result. We will describe the approach in Section 7.3. A CachePOP has one child representing the plan operator tree that created the materialized data set. The plan operator does not induce any source queries. The output schema is inherited from the child operator. The output size of the operator is the size of the cached data set. The cached data is the intersection of cache entries $\bigcap_{e \in E} e.data$ and a possible filter $filter_{cond}$. Hence, the output size of a CachePOP is

$$estSize(p) = \left| \sigma_{filter_{cond}} \left(\bigcap_{e \in E} e.data \right) \right|. \tag{7.7}$$

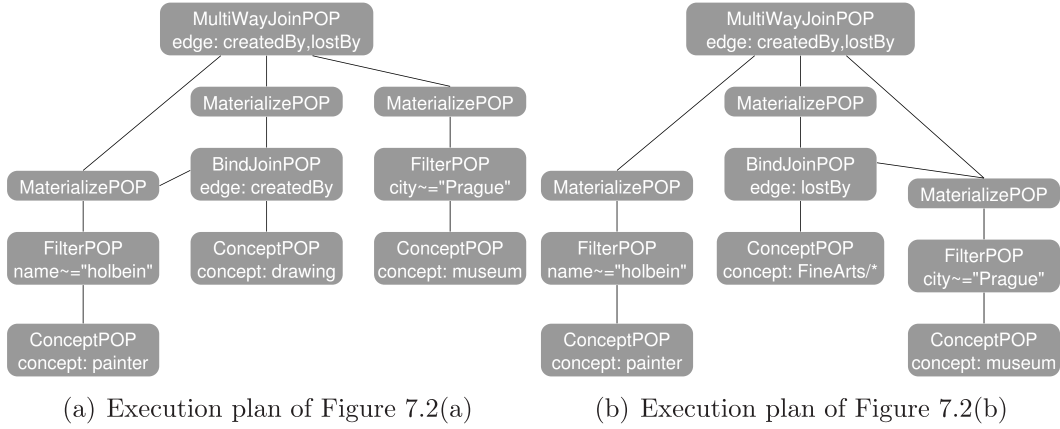
The size $estSize(p)$ is estimated by using general size estimators [SAC+79, Ull90].

Multi-Way-Join Operator. The last plan operator is the *MultiWayJoinPOP*. The inputs of the MultiWayJoinPOP are only MaterializePOP or CachePOP operators. The multi-way join is executed by a global database system on the materialized data. The output schema is the union of all input schemas. The output size is estimated by common join size estimations [SAC+79, Ull90]. The optimal plan is created by the global system and is not considered in the plan optimization in this section.

Plan structure

An execution plan of a materialization query created by a keyword query always has the same structure. The root node is a MultiWayJoinPOP combining all materialized sub-plans. A MaterializePOP is the root of every sub-plan. It materializes the data of exactly one concept in the plan. It uses either a FilterPOP or BindSemiJoinPOP for obtaining the data. A BindSemiJoinPOP has a MaterializePOP and a ConceptPOP (FilterPOP with a free query, respectively) as children. One or more operators can use the materialized result of a MaterializePOP.

Figure 7.4 illustrates two possible plans of the queries in Figure 7.2(a) and 7.2(b).



(a) Execution plan of Figure 7.2(a) (b) Execution plan of Figure 7.2(b)

Figure 7.4.: Execution plans

Basic plan optimization

For the creation of optimal plans regarding the cost functions, we use a dynamic programming algorithm [Kos00, KS00]. Alternative solutions could be iterative dynamic programming or a greedy algorithm [KS00]. However, the number of possible plans is small because of the constraints of the plan structure. The plan structure is constrained, because at first, we always use left-deep bind-semi-join trees with an operator pair consisting of filter and concept operator as left-most leaf, and second we have only one join operator as root that optimizes the join order[3]. Algorithm 7.2 illustrates the approach. Input is a query network (Q, E), and the output is the best plan and its costs. We use a map data structure that consists of a set of pairs (Q', P) with Q' a subset of Q describing the used single concept queries in the set of plans in P. Initially, the algorithm creates the plan operators for single nodes ($createPOP$) resulting in ConceptPOPs or sub-plans consisting of a FilterPOP with one ConceptPOP as child. Specified queries have an additional MaterializePOP as root. Using the dynamic programming approach, for every subset of nodes a plan is created by combining already created sub-plans. During combination, either BindSemiJoinPOP or MultiWayJoinPOP plan operators are created. After every step, the algorithm prunes the plans using the function **prunePlans**. Thereby, only the best plans for every subset are retained. Finally, we find the final set of plans and select the plan with the lowest cost.

In order to decide the best plan, i.e., the plan with the lowest cost, we use the following cost function. We summarize all source queries and transferred source objects in the plan, denoted as $estSrcQ(p)$ and $estSrcObj(p)$ with p the root of the plan. The final costs of the plan are the combination of both values:

$$costs(p) = combine(estSrcQ(p), estSrcQbj(p)). \tag{7.8}$$

For example, a combination function is the weighted sum of both values. In the experiments, we weight both values equally.

[3]We will use a main memory database system as implementation. See Section 8.1

Algorithm 7.2 Dynamic Programming optimization

Input: query network (Q, E)
Output: best plan p
1: **function** OPTIMIZE((Q, E))
2: $plans := \emptyset$ a set of pairs: (Q', P) with $Q' \subseteq Q$ and P a list of plan operators
3: /* *initialize* */
4: **for all** $q \in Q$ **do**
5: add $(\{q\}, \textbf{createPOP}(q))$ to plans
6: **end for**
7: /* *dynamic programming* */
8: **for** $1 \leq i < |Q|$ **do**
9: **for all** $Q' \subset Q$ with $|Q'| = i$ **do**
10: **for all** $(Q'', P) \in plans : Q'' \cap Q' = \emptyset$ **do**
11: $P' := \textbf{createPlans}(plans(Q'), P)$
12: $plans(Q' \cup Q'') := plans(Q' \cup Q'') \cup P'$
13: **end for**
14: **end for**
15: $\textbf{prunePlans}(plans, Q, i + 1)$
16: **end for**
17: let $p \in plans(Q)$ the plan with minimal cost
18: $\textbf{finalize}(p)$
19: **end function**

Plan execution

The result of a query is a set of object tuples that form object networks as an answer to a query network, i.e., valid answers of materialization queries and the keyword query. The plan execution consists of two phases. At first, the system materializes the results of every materialize plan operator and its sub-plan. At second, the materialized tuple sets are joined globally in the second phase. The second phase is simply executed by the global system using common join algorithms like nested-loop, merge, or hash joins. The first phase uses the YACOB mediator for accessing local sources. The materialization sub-plans consist of plan operators of the type ConceptPOP, FilterPOP, and BindSemiJoinPOP. A FilterPOP and its child, a ConceptPOP, are translated into a CQuery statement of the form $\sigma_{cond}(\biguplus_{c \in C} \textbf{ext}(c))$. The systems sends the statement to the YACOB mediator and the mediator returns the set of objects as the result.

Consider the plan in Figure 7.4(a). The system translates the FilterPOP $name \sim=$ "*holbein*" into the statement

$$\sigma_{name \sim= "holbein"}(\textbf{ext}(painter)).$$

The result is materialized. Furthermore, the bind join operation (see Alg. 7.1) uses the result to create queries to the mediator. Initially, the operation projects the input objects to the data property *name* and removes all duplicates. For every value v in this set, the system creates a query $\sigma_{artist=v}(\textbf{ext}(drawing))$ and stores the result objects without duplicates. Subsequently, the third materialization operator is processed, and

the result is materialized, too. Finally, all three materialized object tuple sets are joined using the join operation implemented by a main memory database system.

In this way, we efficiently execute queries over different concept extensions and sources. The approach reuses the Yacob mediator system's capabilities and adds the join functionality. The approach allows the parallel materialization of different filter queries and the usage of efficient joins on the global level. In contrast, an early join of materialized sub-plans might reduce the costs because of early stopping if the system encounters empty results.

7.1.2. Non-Reuse Algorithms

After the discussion of the execution and optimization of separate materialization queries in the previous subsection and the creation of query list networks in Chapter 6, we return to the execution of sets of queries during keyword query processing. The input is a set of query list networks as described in the previous Chapter 6. We require three types of the output (see Chapter 5):

1. the complete results,
2. the k best object networks (i.e., k best non-empty materialization queries), or
3. the best non-empty concept networks with corresponding object networks.

In this section, we shortly summarize two approaches we will adapt to use in our context. The first is the baseline algorithm that creates all query networks and executes them. The second approach uses the query list networks and executes them step-by-step as proposed by Hristidis et al. [HGP03] and Sayyadian et al. [SLDG07]. Both approaches do not reuse intermediate results.

Baseline Algorithm

The baseline algorithm is straightforward and is based on the algorithm presented by Demidova et al. [DZZN09]. First, we take all query list networks. From every query list network, the algorithm builds all valid materialization queries. Every materialization query has a score. Depending on the required answer set, we propose the following three algorithm alternatives:

Top-k materialization queries: the top-k version of the algorithm sorts the queries by descending score order. After this first step, the algorithm executes query by query until k queries were successfully executed.

All object networks: in the first step, the algorithm removes all query network duplicates. Duplicates can occur if a keyword acts as schema keyword. After the first preprocessing step, all queries are executed.

Top-k concept networks: in this case, we group all queries by the corresponding concept network and sort the queries in the groups by descending score. Furthermore, we sort all groups by the best query score in the group. Now, we

execute the queries step by step and use always the best global query. If a concept network group contains a successful query, we will output this network with all queries and the successful query result. The algorithm continues until k results are found or all queries are executed.

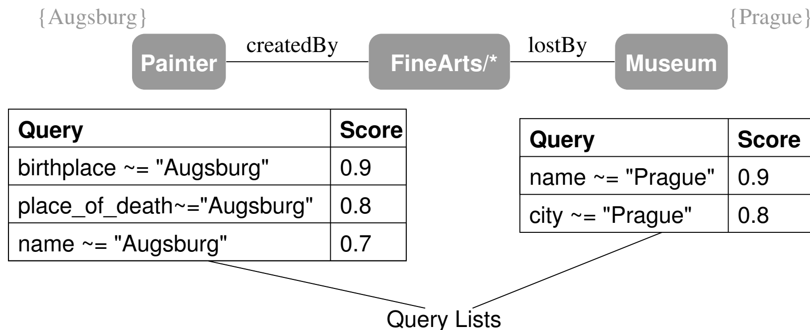
Query List Network based Algorithms

In this section, we describe the second class of algorithms that directly execute query list networks. These algorithms are adapted from the algorithms proposed by Hristidis et al. [HGP03] and Sayyadian et al. [SLDG07]. The basic idea is the exploitation of the monotonicity of the ranking function. The monotonicity expresses itself as follows. Considering two query networks QN_1 and QN_2 corresponding to the same concept network. For all but one ($q_{1i} \in QN_1(Q)$ and $q_{2j} \in QN_2(Q)$) single concept queries, the queries have the same score. The ranking function is monotone, if

$$score(QN_1) > score(QN_2) \Leftrightarrow score(q_{1i}) > score(q_{2j}).$$

Given these characteristics, we execute query list networks as follows.

The algorithm always selects the query list network with the highest possible score. The highest possible score is the highest score of a query network, which was not yet constructed from the query list network. The highest scoring query list network is executed as follows. Consider the query list network illustrated in Figure 7.5 as example.



Query	Score
birthplace ~= "Augsburg"	0.9
place_of_death~="Augsburg"	0.8
name ~= "Augsburg"	0.7

Query	Score
name ~= "Prague"	0.9
city ~= "Prague"	0.8

Figure 7.5.: Query list network

Initially, the algorithm selects the first two entries of the query lists and creates a query network. The network is executed. If the query has been successful, we add the query and its result set to output. In the case of top-k concept network semantics, we add the complete network and all related networks to the output. Otherwise, we mark the top queries as used (Figure 7.6(a)). The algorithm recomputes the new maximal possible score of the network using the top query network that contains at least one non-used single concept query. In the following, we select the best network again. If we assume, the next best network is the same network, the algorithm selects the most promising single concept query, marks it as used, and enumerates all query networks together with the used single concept queries of the second list (see Figure 7.6(b)). The algorithm executes the queries if they have a higher score than the k-th result in the

output. We execute the queries separately because we assume that only conjunctive conditions are allowed as source queries. The algorithm stops if the k-th result has a higher or equal score to the best existing query list network or all queries are exhausted.

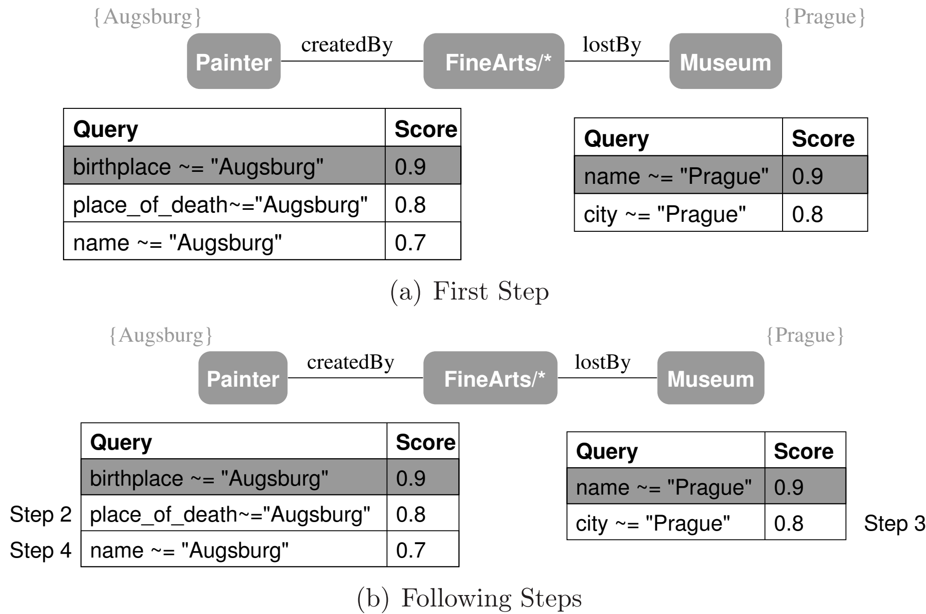


Figure 7.6.: Query list network processing

This approach is especially successfully used for top-k queries [HGP03, SLDG07], either for object networks or concept networks. Generating all results the approach is extremely expensive. In our setting of restricted query capabilities and external sources, the approach must be adapted to reuse information of intermediate and previous results during query processing.

7.1.3. Motivation for Query Re-Using Algorithms

As we have seen in the previous sections, the processing of concept-based queries uses expensive operators like the bind-join operation. The system generates many source queries and transfers many objects During processing. Analyzing the queries, we can make the observation that many queries overlap. Furthermore, many intermediate queries are empty. Thus, there is a significant potential of re-using intermediate or previous results during processing to reduce the query costs.

For example, we tested keyword queries for the IMDB test database (see Section 8.2.2). We created and analyzed for keyword queries of the size 2 to 5 as well as for a maximum query network size of 3 and 4 the materialization queries. The results are presented in the Figures 7.7. The figures show that the number of materialization queries and query networks grows with increasing parameter values. Also, the number of average and maximum re-usage of single concept queries increases. We define a join in this motivation as one join with a single concept query. We do not consider longer join paths here. Also, the reuse of joins increases. In Table 7.3, we show the percentage

of single concept queries and simple joins that occur in more than one materialization query. We present the data with respect to the keyword query size and the maximum network size $size_{max}$. The results show a high percentage of reuse in the queries.

	Number of keywords			
$size_{max}$	2	3	4	5
3	72.7%/61.1%	92.8%/83.2%	93.8%/76.2%	98.6%/79.1%
4	80.9%/92.6%	94.6%/95.9%	95.3%/99.3%	98.8%/99.5%

Table 7.3.: Percentage of overlapping query parts (single concept queries/joins)

In order to minimize the costs, we investigate the following points. First, we try to detect empty results using some statistics. A similar approach is also used in Demidova et al. [DZZN09] and Chai et al. [CBC+09]. We will discuss the differences in the end of the chapter in Section 7.7.

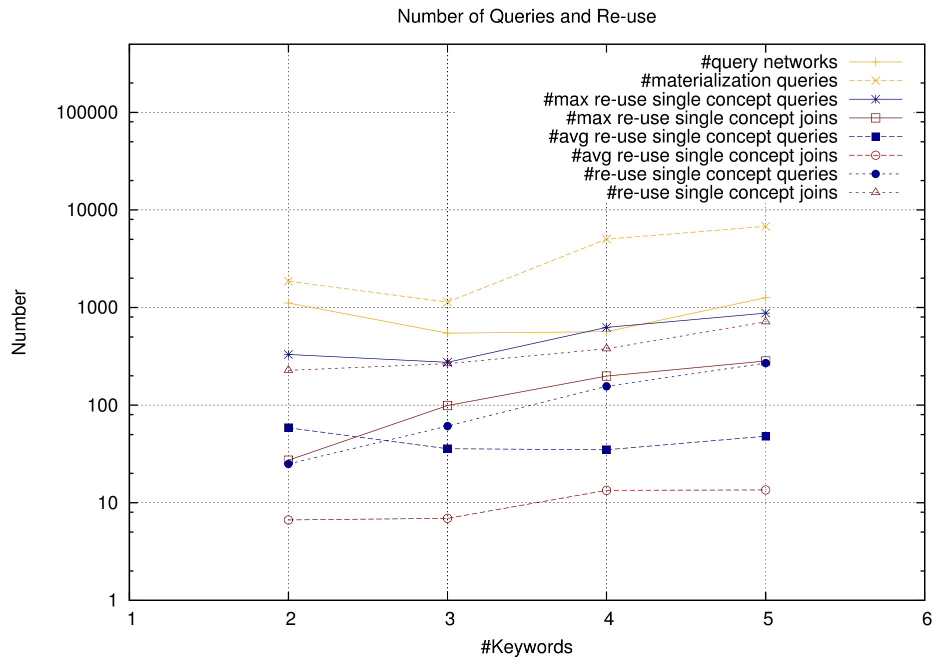
In the second case, we reuse actual intermediate results. That is necessary to minimize the access to the local sources. We distinguish two approaches. First, it is possible to cache results of separately executed queries and second to optimize multiple queries together to find an optimal global plan. For example, Agrawal et al. [ACD02] proposed to use multi-query optimization in keyword search in relational databases but did not provide a solution. Hristidis et al. [HP02] proposed a greedy algorithm to find a globally near-optimal plan for getting all results. We will discuss how to optimize query list networks and a complete keyword query, i.e., a set of similar materialization queries. Markowetz et al. [MYP09] and Li et al. [QYC11] proposed multi-query data structures for processing keyword search over data streams. Our approach also supports top-k queries as well as materialization queries instead of tuples. In the following section, we provide solutions to avoid re-computation of empty results and reuse of successful intermediate queries, respectively.

7.2. Detection of Empty Results

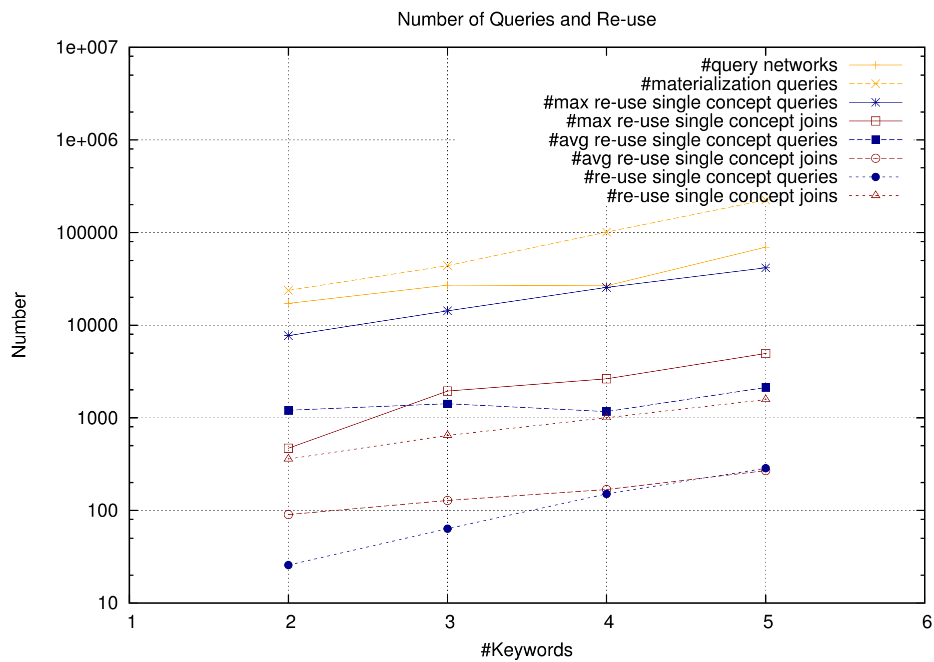
The first step of optimization during query execution is to avoid re-computation of empty results. Empty results emerge because of missing join partners or combination of keywords in single concept queries that cannot be satisfied. For example, the index finds the keywords “holbein” and “gogh” in the artist concept. One corresponding, empty query is $artist/name \sim= \text{"}holbein\text{"} \wedge artist/name \sim= \text{"}gogh\text{"}$. As parts of concept-based queries are often shared during execution of keyword queries, we argue that we can significantly improve the query performance by storing empty results and avoid their re-computation.

Generated concept-based queries consist of selection and join operations, which propagate empty results [Luo06]. That means that these operators will have an empty result if one input of these operators is empty. Therefore, the execution of the complete plan can stop if at least one sub-query has had an empty result.

The problem of detecting empty-result queries consists of the following sub-problems:



(a) Number of networks and queries for $size_{max} = 3$



(b) Number of networks and queries for $size_{max} = 4$

Figure 7.7.: Query overlapping

Concept-based Query coverage: A query q covers a second query q' if q' query always has an empty result if query q has been empty. To maximize the usage of empty result description, we have to exploit query coverage in concept-based queries, e.g., sub-concept relationships.

Data structure for empty-result descriptions: The system has to store queries that had an empty result. We have to encode the queries in a way, that the re-usage is maximized and efficient. Furthermore, the data structure has to consider constraints like limited storage size.

Statistics management: As the storage space is limited and statistics can be outdated by changes in the data sources, we have to implement replacement strategies as well as a statistics aging approach.

Use in keyword query processing: Using the detected empty results, the system has to decide to execute or not to execute a query or a query set.

7.2.1. Concept-based Query Coverage

In this section, we describe how to use of empty-result information by introducing the empty query coverage for concept-based queries in the context of the YACOB system. Let S be a concept schema and $D(S)$ a global, virtually integrated database instance of S. Then, we define *empty query coverage* in this context as follows.

Definition 7.1 Query Coverage For Empty Results
Given two concept-based queries q and q' over S. For any instance $D(S)$, a query q covers q' if q has an empty result in $D(S)$, i.e., $R(q, D(S)) = \emptyset$, then q' is also empty in $D(S)$. We denote query coverage as $q' \sqsubseteq q$. We say q' is covered by q, i.e., $q' \sqsubseteq q$.
□

The query coverage is thereby different to query containment as described in the following Section 7.3, in which we compare query coverage and query containment. To decide coverage for generated concept-based conjunctive queries, we define the following set of coverage rules. The discussion starts with single concept queries. Subsequently, we extend these rules to query networks.

A single concept query has the form $\sigma_{cond}(\mathbf{ext}(c))$ or is extended to $\sigma_{cond}(\biguplus_{c \in \mathcal{C}} \mathbf{ext}(c))$. That means, the queries consist of a conjunctive selection *cond* over a union of concept extensions. The selection condition *cond* is a conjunction of single predicates of the form $p \sim= kw$ (keyword containment), $p = k$ (category selection), or exists(p) (test for existence of an attribute value). We define the following rules for single concept queries.

In *rule 1*, we compare the two types of predicates, keyword containment and existence test. If the existence test is false, i.e., the value is null, the keyword containment predicate will be false, too. Thus, *rule 1* states

$$\sigma_{p \sim= kw}(\mathbf{ext}(c)) \sqsubseteq \sigma_{\text{exists}(p)}(\mathbf{ext}(c)).$$

We also say, the predicate exists(p) covers $p \sim= kw$. Equivalently, the predicate exists(p) also covers $p = k$ with k a category.

The second rule deals with a complete conjunctive condition. There are two observations. First, additional predicates in a conjunctive condition will restrict the result further. Second, if we exchange an exists predicate by a keyword containment predicate on the same property, the result set will also be restricted further. As an empty result of a less restricted query induces an empty result of a more restricted query, we can define *rule 2* as follows: if all predicates in condition *cond* cover a predicate in *cond'*.

$$\sigma_{cond'}(\mathbf{ext}(c)) \sqsubseteq \sigma_{cond}(\mathbf{ext}(c)).$$

That means, all predicates of *cond* also occur in *cond'* or the predicates have a covered counterpart in *cond'*. The condition *cond'* is allowed to contain further predicates. In this case, we also say, the condition *cond* covers *cond'*.

A single concept query is also allowed to use a set of concepts in one concept hierarchy. The following *rule 3* deals with such kind of queries. Assuming the condition *cond* covers *cond'*, the rule states

$$\sigma_{cond'}\left(\biguplus_{c \in C'} \mathbf{ext}(c)\right) \sqsubseteq \sigma_{cond}\left(\biguplus_{c \in C} \mathbf{ext}(c)\right) \text{ if } C' \subseteq C.$$

That means that we have a covering query for every concept $c \in C'$ following the rules 1 and 2. A special case of rule 3 is

$$\sigma_{cond'}\left(\biguplus_{c \in \Phi^{+}_{\mathbf{is_a}}(c'')} \mathbf{ext}(c)\right) \sqsubseteq \sigma_{cond}\left(\biguplus_{c \in \Phi^{+}_{\mathbf{is_a}}(c')} \mathbf{ext}(c)\right) \text{ if } c' \text{ is super-concept of } c''.$$

The expression $\Phi^{+}_{\mathbf{is_a}}(c')$ computes all direct and indirect subconcepts of c'. Thus, we can apply the rule 3 for this case.

Rules 1-3 deal with single concept query coverage. Now, we extend empty result coverage for query networks. A query network (Q, E) consists of single concept queries and edges between them. Edges are inter-concept properties and describe the join conditions between the results of the single concept queries. That means, a network (Q, E) corresponds to a conjunctive SPJ query.

Assume two query networks (Q, E) and (Q', E'). We define *rule 4* as

$$(Q', E') \sqsubseteq (Q, E),$$

i.e., (Q', E') is covered by (Q, E), if

1. for each query $q \in Q$ exists at least one query $q' \in Q'$ and q' is covered by q', i.e., $q' \sqsubseteq q$, and

2. for each edge $(q_i, q_j, p) \in E$ exists one edge $(q'_i, q'_j, p) \in E'$ with $q'_i \sqsubseteq q_i$ and $q'_j \sqsubseteq q_j$.

We can proof this rule as follows. In the initial step, we assume two query networks with only one query, i.e., $q = (\{q_1\}, \emptyset)$ and $q' = (\{q_1'\}, \emptyset)$. We use rules 1 to 3 for deciding if q_1 covers q_1' by testing the covering relationship between q_1 and q_1'. We now extend it to query networks. Let $q_1' = (Q_1 \setminus \{q_{1n}\}, E_1 \setminus \{(q_{1i}, q_{1n}, p)\})$ and $q_2' = (Q_2 \setminus \{q_{2m}\}, E \setminus \{(q_{2j}, q_{2m}, p)\})$ be two query networks with $q_1' \sqsupseteq q_2'$. Then, it holds $q_1 \sqsupseteq q_2$ if $q_{1n} \sqsupseteq q_{2m}$ and $q_{li} \sqsupseteq q_{2j}$. This condition leads to the rule 4.

7.2.2. Data Structure

As we will show below, the system checks query coverage during optimization. This data coverage check requires an efficient data structure to decide if a query is covered by another query that is known to have an empty result. For that, we describe the data structure for holding information about empty query results. The structure has to efficiently store the queries as well as it has to allow to efficiently test for empty query coverage during optimization. We represent concept-based queries using query networks and not physical operator trees. The data structure comprises three levels (see Figure 7.8). At first level, the system checks the size of the query (number of single concept queries). Thereby, only a query of the same size or smaller size can cover another query. The second level contains the set of concepts in the query. Only if the concepts cover a query to be tested, the following level has to be checked. There are different queries for every concept set. Therefore, the third level is a list of query representations for every concept set. The data structure follows the idea of Luo for relational databases [Luo06], and extends it by the size level.

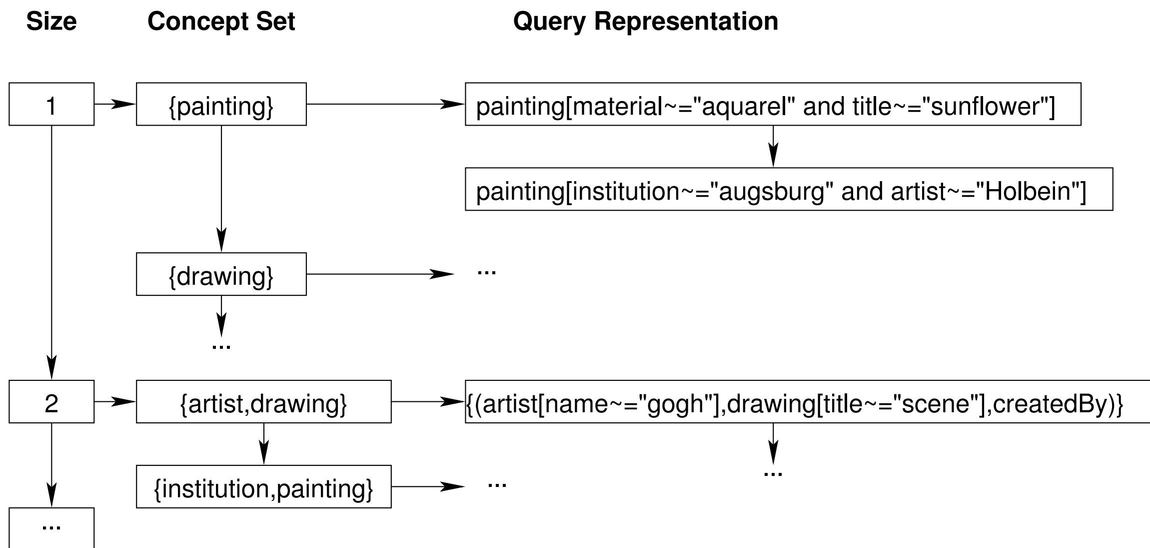
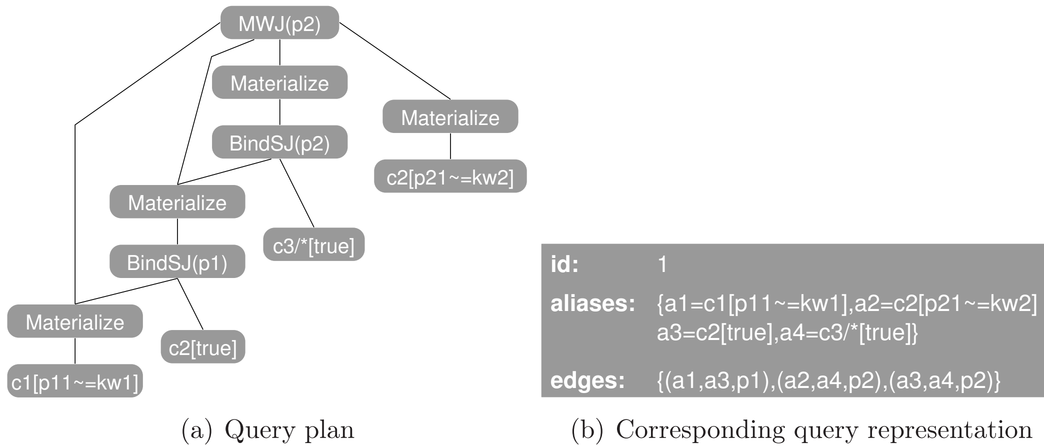


Figure 7.8.: Empty result set detection

A query representation consists of a list of aliases and a list of edges. An alias consists of the concept name, an indicator whether sub-concepts are included, and a possible selection condition. The aliases are ordered by their coverage of their conditions, i.e., covered aliases are at the beginning of the list. The order ensures that we will find covered results as we will describe below. That also means that a concept can occur

several times in the alias representation, if a free query to a concept is used more than once in the query. An edge consists of two aliases and the edge name. The edges are ordered by the position of the aliases. Furthermore, every query representation has an identifier.

Assume the query plan in Figure 7.9(a). The plan consists of two specified and two free queries connected by three joins. We illustrate the corresponding query representation in Figure 7.9(b). It has four aliases, the queries, ordered by covering and the list of edges as a representation of the join tree. A query representation can be seen as a normalized notation of a query tree. In order to test query coverage efficiently, we exchange the concept name by the corresponding Dewey identifier of the concept. The system checks fast sub-concept relationships by comparing prefixes of the Dewey identifiers.



(a) Query plan

(b) Corresponding query representation

Figure 7.9.: Empty result representation

7.2.3. Empty Result Statistics Management

We now discuss the management of the empty result statistics. The management comprises the questions: how and when to add empty results, how many entries to store, and when to remove entries.

The system discovers empty results during query processing. After every FilterPOP or BindJoinPOP execution, the query processor tests if the result is empty. If it is empty, the system adds the corresponding query representation to the empty result statistics. The query representation was created during optimization and is used to insert it into the statistics. In the case of the MultiWayJoinPOP operation, we first test all possible join combinations for empty results and then compute the complete join. This approach was proposed by Demidova et al. [ZZDN08, DZZN09] and complements our checks. This additional check is acceptable, because queries to materialized data are not expensive.

We have to consider three points in statistics management: first, the statistics has a limited size, second, underlying sources might change over the time, thus, empty results can be non-empty after a certain time, and third: a newly inserted empty result might cover other entries in the statistics.

We limit the number of entries to a pre-configured $maxSize_{empty}$. If this number of entries is exceeded, the system will remove an old entry. Our system implements the least recently used (LRU) strategy [SSH05]. For this task, we use an LRU stack implementation. Other replacement algorithms can be used, too. For example, Luo used the DCLOCK approach [Luo06].

As sources can change, the system limits the time of an entry in the cache. Every entry has a creation time stamp *createtime*. Furthermore, we configure a maximal keep time *keeptime*. If the difference between the current time and the creation time is larger than the *keeptime*, the system will remove the entry from the statistics. The system checks the time during accessing an entry. The system will recognize the result only as empty, if the entry is not older than the keep time. Otherwise, the system will remove the entry. Unused old entries are eventually removed by the LRU approach. In this way, the statistics are kept up-to-date.

In some cases, a new entry might cover existing results. As this situation is seldom, we do not check for covered entries. The check for covered entries is expensive, because it involves all empty result descriptions with a size bigger and equal to the size of the new query. Covered results will eventually be removed by one of the former two checks because they will not be used anymore.

7.2.4. Use of Detected Empty Results during Keyword Query Processing

The set of generated queries is processed step by step. Thereby, later queries reuse portions of queries executed before. The system uses the proposed data structure for checking empty results during optimization. For the check, we modify the dynamic programming Algorithm 7.2 on page 145 as follows.

In every step of the algorithm, we test if the sub-plans created in this step occur in the empty result statistics. If any sub-plan would create an empty result, the optimization stops and returns an empty plan with the information the result is empty. This is possible because all operations propagate empty results. During the optimization, we will create the query representation for every sub-plan by merging the representation of the combined plans.

The check for emptiness is executed as follows. Given is a sub-plan of a given size (number of aliases). In the first step, we check whether there empty result entries in the statistics with this size. The algorithm has to check only query representations of the same size, because smaller sub-queries have been checked in a previous step of the optimization and larger queries cannot cover the query. The second step uses the selected list of used concepts. If the used concept set covers the queries concept set, the algorithm will proceed to check the complete query representations containing the selected concept set. In this case, all conditions and join conditions are checked for coverage. If a covering representation has been found, we know the sub-plan creates an empty result, and the optimization, and subsequently, the query processing stops.

Using the information about empty intermediate results improves the query execution because many empty results are common in materialization queries created

by a keyword query. In order to validate this, we evaluated the performance. The evaluation results are presented in Section 8.3.

7.3. Re-Using Intermediate Results

After detection of empty results, we now present an approach to reuse materialized intermediate query results. The basic idea is to materialize intermediate results in a cache, execute every materialization query separately but reuse the stored, shared results. That kind of cache is related to the semantic cache approach [DFJ+96, Sel88b, SSV96, WA10, AKS99, IKNG10, Kar03]. In particular, the reuse of shared results is vital in heterogeneous, distributed systems because of high object transfer time, high query execution costs, or billing costs [ACPS96, AKS99]. Furthermore, limited query capabilities of sources even increase this problem because they require operations like the bind join operation.

In this section, we describe our semantic cache-based approach that is used to reduce the number of source queries during keyword query processing. Initially, we describe how we organize materialized intermediate results. The results are represented as conjunctive queries. The selection conditions consist of constant selections ($p \sim= kw$ and $p = k$) and existences tests exists(p). Furthermore, we represent an intermediate result as a subset of a concept. We denote this as derived concept. We define the access to the cache, the query containment in the concept-based case, and the retrieval of cache entries based on different query match types. The third part of this section describes the basic use of cached intermediate results during optimization of materialization queries. The algorithm determines when to materialize results. We conclude the section with the description of cache eviction strategies. Updates of the cache are not discussed, rather cache entries are invalidated after a maximum keeping time. We start with the presentation of the cache structure.

7.3.1. Materialized Results Organization and Access

Given the previous discussion, we discuss the structure of the cache in the following dimensions.

Caching level: In the mediator system architecture, two caching levels exist: on wrapper level and mediator level [LC99]. Translated to the concept-based mediator scenario the possible caching strategies are either by source or concept [KSGH03, Kar03]. In the first approach, the cache stores objects by source and allows a fine granular storage of local XML objects on wrapper level. The second approach stores global objects per concept. Thus, every query to a concept can create a cache entry for a concept [KSGH03, Kar03]. In the case of keyword queries, the user expects globally connected concepts as results. As the connections are defined globally, the concept level approach is the feasible solution.

Cache content: The second decision focuses on the content of the cache. There are two possibilities, either the results of joins, i.e., object tuples, or only the

necessary objects to compute globally the join. In the first case, we always store the join results, i.e., object tuples. However, if we do this for a join sequence, the system stores redundant data. In the second case, we try to replace source accessing operations by global operations. Therefore, we store derived concepts following the idea of derived horizontal relational fragmentation [ÖV11] and the idea of derived concepts [ABFS02b]. This approach avoids redundant storage of results and complements the semi-join query processing strategy presented in Section 7.1.1. Thus, we store results of FilterPOP and BindSemiJoinPOP executions. In consequence, global join operations have to be re-computed.

Query containment: In order to use the materialized results, we have to define query containment and supported match types. As we use the materialized results only for keyword search, we can reduce the cache to basic constant comparison predicates. Furthermore, the queries are conjunctive and use sets of concepts or single concepts. The query containment check will be complementary to the empty result covering described in Section 7.2. Different matching types exist between a query and a cache entry. We have to evaluate, which we will support and how compensating queries are constructed.

Eviction strategy: The last point comprises the management of the cache. In order to keep the cache up-to-date, we will evict entries after a given timespan. In the case of a limited cache pool, we describe a cache entry eviction strategy.

Derived concepts and derived object sets

In order to explain the cache structure and the cached queries, we describe derived concepts that represent results of selection and semi-join queries.

The definition of derived concepts is based on the idea of Amann et al. [ABFS02b]. Let $p = cp_1/cp_2/\ldots/cp_n$ be a sequence of concept properties denoted as *concept property path*. For all $1 \leq i < n$, it holds $cp_i.dest = cp_{i+1}.src$. Given a concept $c \in \mathcal{C}$, a *derived concept* is c/p with $cp_1.src = c$.[4] This means, the derived concept contains all objects of concept $cp_n.dest$ that can be reached from any object in c through p. As the concept property path p is allowed to be empty, the concept c is also a derived concept. We now extend the derived concept definition to derived object sets.

Let $lp = name_1[cond_1]/name_2[cond_2]/\ldots/name_n[cond_n]$ be a sequence of labels $name_i$ that correspond to concept properties denoted as *label path*. A condition $cond_i$ represents a selection of concepts from a concept schema. In particular, the condition is either $name =$ *"conceptname"* or $name =$ *"conceptname"/**. The former selects a concept with the given name, and the latter also selects all sub-concepts. The set of concepts created by $cond_i$ is denoted as C_i. A label path lp represents a set of concept property paths. A concept path $p = cp_1/cp_2/\ldots/cp_n$ conforms to lp if

1. $cp_i.name = name_i$, for $1 \leq i \leq n$ and
2. $cp_i.dest \in C_i$ and $cp_{i+1}.src \in C_i$ for $1 \leq i < n$.

[4] A concept property cp_i can occur as original property or as inverse property in the path.

Let $C_0 \subset \mathcal{C}$ be a set of concepts then C_0/lp denotes a *derived concept set*. The concept set C_0/lp represents all derived concepts c/p with $c \in C_0$ and p conforms to lp.

Example 7.1 *For example, let $C_0 = institution/*$ represent the concept "institution" and all of its sub-concepts. Then, the derived concept set $C_0/lostBy[name = "Drawing"]/createdBy$ would represent all persons who created a drawing that was lost by an institution because*

1. *$lostBy[name = "Drawing"]$ selects all concept properties between an institution and the concept "drawing",*

2. *createdBy selects all persons that are reached from a drawing instance.*

The extension of a derived concept set is denoted as *derived object set*. It is the union of the extensions of all represented derived concepts. Using the join rewriting of concept properties, a derived object set for C_0/lp is defined as

$$\mathbf{ext}(C_0/lp) = \mathbf{ext}(C_0) \rtimes_{JM(name_1)} \mathbf{ext}(C_1) \rtimes_{JM(name_2)} \dots \rtimes_{JM(name_n)} \mathbf{ext}(C_n) \quad (7.9)$$

with $\mathbf{ext}(C_i) = \biguplus_{c \in C_i}(\mathbf{ext}(c))$ for $1 \leq i \leq n$. The use of the semi-join $\rtimes$ directly computes the extensions of the derived concept. In consequence, together with filter conditions, a derived object set directly describes the materialized intermediate results in the query processing.

Example 7.2 *Consider the example in Figure 7.4(a) (see page 144) containing the sub-plan: $p' = \sigma_{name\sim="holbein"}(\boldsymbol{ext}(painter))$ and $d' = p' \rtimes_{JM(createdBy)} \boldsymbol{ext}(drawing)$ to compute all drawings that were created by "Holbein". The derived object set d' is described by derived concept set $painter/createdBy[name = "drawing"]$ and the selection condition on the "painter" concept.*

Cache structure

The cache stores materialized semi-join results and provides the results to further queries. The structure follows the idea of the Yacob internal cache [Kar03, KSGH03]. While the Yacob cache organizes the entries first by concepts and then by selection conditions, the keyword query cache is organized by derived concept sets.

Initially, the cache is partitioned by concepts of the concept schema. For every concept, we store a list of derived concept sets that represent a subset of the concept extension or the deep concept extension. That means that the final concept of the label path is either the concept with all sub-concepts or the concept alone. Every entry in the derived concept list is of the form `startConcept[/*]/labelpath`. The label `startConcept` represents the starting concept or with all sub-concepts `[/*]`. The label path is used as defined. The derived concept sets contain a list of cache entries. The data of a cache entry is a subset of the extension of the derived concept set, i.e., the derived object set. If the `labelpath` is empty, the derived concept set entry represents the concept extension of `startConcept` or its deep extension, respectively.

Every cache entry represents a semi-join result. A cache entry $e = (id, cond, metadata, data)$ comprises a unique identifier id, a condition $e.cond$, meta

data *e.metadata*, and a pointer to the actual data *e.data*. As every semi-join result is derived from a specified query, i.e., a selection on the starting concept, every cache entry is specified by a conjunctive *cond* of keyword containment or category selection predicates. These predicates refer all to the starting concept. Furthermore, the condition *cond* also contains exists(p) predicates that can refer to any of the concepts sets in the label path, because they can be part of a free query. The condition is represented by a set of the contained predicates.

The meta data of a cache entry is used for cache management and during optimizations of queries. The meta data $e.metadata = (createtime, lastAccess, size, costs, hits)$ consists of the creation time $createTime$, the time of the last access $lastAccess$, the size of the represented data set $size$, the costs to create the data $costs$, and the number of usages $hits$. We access these values by the notion $e.metadata.x$.

Example 7.3 *We illustrate the cache structure in Figure 7.10. Firstly, the cache is partitioned by concepts, e.g., book, graphics, and painter. They are derived concepts without label path. Secondly, the derived concept sets "Painter/createdBy[/*]" and "Museum/lostBy[name="paintings"]/createdBy" are added as sub-entries of "graphics" and "painter", respectively. Note, in the case of Painter/createdBy[/*], the concept property "createdBy" is used inversely. Furthermore, the destination concept is determined by the parent concept and is modified by [/*] to include sub-concepts. Every derived concept set entry contains a list of cache entries. For example, cache entry 3 represents the data* $\sigma_{name \sim = "holbein"}$ $\textbf{\textit{ext}}(Painter) \rtimes_{JM(createdBy)} \biguplus_{c \in \phi^{+}_{\underline{is_a}}(Graphics)} \textbf{\textit{ext}}(c)$. *Cache entry 5 holds the data* $\sigma_{city \sim = "prague"}$ $\textbf{\textit{ext}}(Museum) \rtimes_{JM(lostBy)} \textbf{\textit{ext}}(Paintings) \rtimes_{JM(createdBy)}$ $\textbf{\textit{ext}}(Painter)$.

In the following, we describe two distinct cases of query networks. In keyword queries, we often have a partial query tree as depicted in Figure 7.11(a). Using semi-joins, we get the plan $b' = \sigma_{title \sim = "flowers"} \textbf{ext}(book)$, $a' = b' \rtimes_{writtenBy} \textbf{ext}(author)$, and $i' = b' \rtimes_{lostBy} \textbf{ext}(institution)$. The results b', a', and i' would be materialized. A second example is illustrated in Figure 7.11(b). Here, the partial network contains a central node connected to two concept nodes of the same kind. The corresponding query plan would include $b' = \sigma_{title \sim = "flowers"} \textbf{ext}(book)$ and $a' = b' \rtimes_{writtenBy} \textbf{ext}(author)$. The result a' is used twice in the join plan with different aliases.

In contrast to the work of Karnstedt et al. [Kar03, KSGH03], this data structure does not ensure the disjointedness between cache entries. Therefore, we have to find the best matching entries. We will describe the corresponding lookup algorithms after the query containment definition.

Query containment

For a given query, the system tries to answer it using previously materialized query results. In order to select and use data in the semantic cache, a query containment check is necessary [Sel88a, DFJ+96, KB96]. Every cache entry represents the result of a query. The cache entry would provide objects to the given query, if the cache

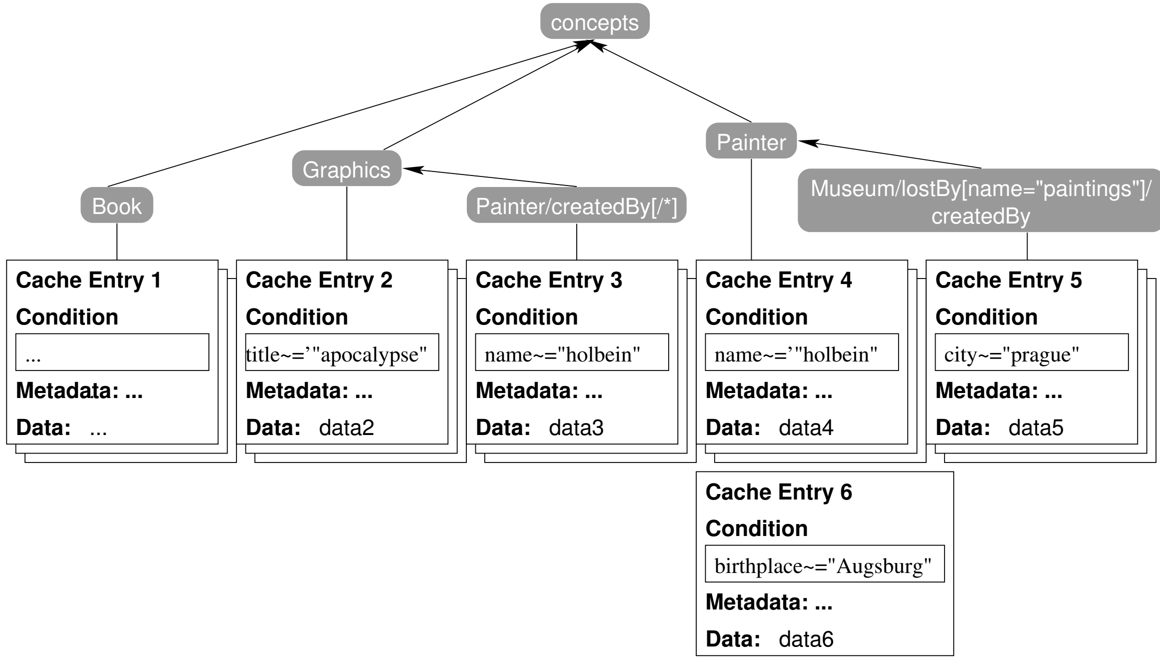
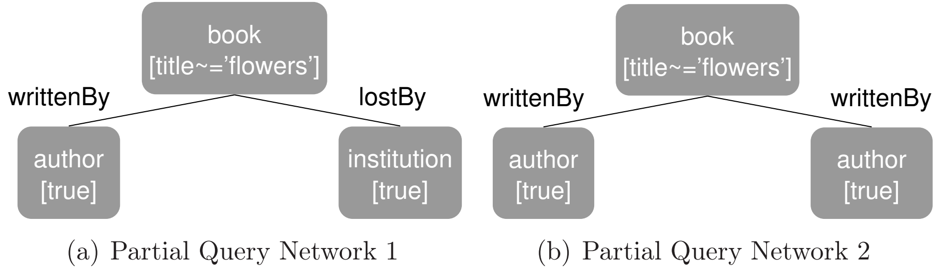


Figure 7.10.: Cache structure



(a) Partial Query Network 1 (b) Partial Query Network 2

Figure 7.11.: Exemplary distinct cases

entry overlapped the result set at least. Query containment is based on query satisfiability [GSW96]. Let S be a concept schema and $D(S)$ a global instance of S. Query containment is defined as follows [Ull90]:

Definition 7.2 Query containment
Let q and q' be two queries over S with the result sets $R(q, D(S))$ and $R(q', D(S))$ for a database instance $D(S)$. A query q is contained by q', $q \subseteq q'$, if for all databases instances $D(S)$ $R(q, D(S)) \subseteq R(q', D(S))$. In the case $q \subseteq q'$ and $q' \subseteq q$, q and q' are equal denoted as $q \equiv q'$. □

The relationship between query coverage (see Section 7.2.1) and query containment is expressed as follows. Assume two queries in query containment relationship $q \subseteq q'$. Assume a database instance $D(S)$ where $R(q', D(S)) = \emptyset$. It follows $R(q, D(S)) = \emptyset$ because $R(q, D(S)) \subseteq R(q', D(S))$ for all $D(S)$. Therefore, it holds, $q \subseteq q' \rightarrow q \sqsubseteq q'$. The opposite direction, $q \sqsubseteq q' \rightarrow q \subseteq q'$, is not true. For example, assume the query q' and $q = q' \bowtie C$. Assume a database instance $D(S)$, where $R(q', D(S)) = \emptyset$. It follows

that $R(q, D(S)) = \emptyset$ because $\emptyset \bowtie C = \emptyset$. Thus, $q \sqsubseteq q'$. On the other hand, assume a database instance $D'(S)$ with $R(q', D'(S)) \neq \emptyset$. In this case, the result $R(q', D'(S))$ is not a subset of $R(q', D'(S))$.

Different *matching types* exist between two queries. Table 7.4 describes the possible matching types [LC01, KSGH03]. Let q be a query and $q(e)$ be a query represented by a cache entry e. We can distinguish five match types. In the first case, the query of the cache entry $q(e)$ equals to the query q. The query processor can directly use the cache entry data. An additional filter and a complementary query are not necessary. The second case states that $q(e)$ contains the query q, i.e., the cache data set is a super-set of the query result. In this case, we do not need a compensating query, but have to filter the cache entry set. The third case is $q(e) \subseteq q$, which means that the cache provides only a part of the result set of q. Therefore, the system computes a compensating query that only retrieves the additionally needed objects. The compensating query is $q \wedge \neg q(e)$. The cache content can also overlap the query result set. That means that we need to filter the cache content to get the part used in q as well as the system has to create a complementary query to retrieve the remaining part of the result set. Finally, the disjointedness between cache entry and query means that the cache cannot contribute to the query result at all.

Match type $(q, q(e))$	**Description**	**Cache part of result data**	**Complementary query**
Exact	data in q and $q(e)$ identical	$e.data$	none
Containing	$q(e)$ contains q	q of $e.data$	none
Contained	$q(e)$ contained by q	$e.data$	$q \wedge \neg q(e)$
Overlapping	q and $q(e)$ overlap	q of $e.data$	$q \wedge \neg q(e)$
Disjoint	$q(e)$ not part of q	none	q

Table 7.4.: Match types in the semantic cache [KSGH03, LC01]

In this thesis, we focus on the match types "Exact" and "Containing". The tested queries are semi-join left-deep trees. A query q is represented by a derived concept set $C_q/labelpath_q$ and a conjunctive condition $cond_q$. A cache entry e is determined by its derived concept set $C_e/labelpath_e$ and the cache entry condition $cond_e$. The match type between query and cache entry is computed in two steps: (i) containment of the derived concept sets and (ii) containment of the conjunctive conditions.

The a derived concept set $C_e/labelpath_e$ contains($\supseteq$) $C_q/labelpath_q$ if

1. $C_e \supseteq C_q$,

2. $name_{ei} = name_{qi}$ with $name_{ei} \in labelpath_e$ and $name_{qi} \in labelpath_q$, and

3. $C_{ei} \supseteq C_{qi}$ with C_{ei} (C_{qi}) the target concept sets of the concept property $name_{ei}$ ($name_{qi}$), for $1 \leq i \leq n$.

The concept sets $C_e/labelpath_e$ and $C_q/labelpath_q$ are equal, iff $C_e/labelpath_e \supseteq C_q/labelpath_q$ and $C_q/labelpath_q \supseteq C_e/labelpath_e$. If C_{en} is a superset of C_{qn}, we

create a filter condition $cond_{dc} = \{concept \in C_{qn}\}$, which ensures that only objects of the concepts C_{qn} are returned.

Example 7.4 *Consider the concept sets* $dc_e = (Artist/*)/createdBy[name = "paintings"/*]$ *and* $dc_q = Painter/createdBy[name = "paintings"]$*. It holds* $dc_e \supseteq dc_q$ *because firstly,* $(Artist/*)$ *denotes the concept Artist and all sub-concepts, which include the concept Painter (rule 1). Secondly, both label paths are equal (same concept property labels and target concept sets). The concept sets are not equal. The filter condition is* $concept \in \{paintings\}$ *(rules 2 and 3).*

If the concept sets have a containment relationship, we will compare $cond_q$ and $cond_e$. The containment test does not have to consider range queries because the condition contains only comparisons with constants. A predicate $pred_e$ contains a predicate $pred_q$ if

1. they refer to the same concept in the derived concept set,
2. $pred_e = pred_q$, or
3. $pred_e$ covers $pred_q$, that means, $pred_e = exists(p)$ and $pred_q$ is either $p \sim= kw$ or $p = k$.

A condition $cond_e$ contains a second condition $cond_q$ if we have found for every predicate $pred_e \in cond_e$ at least one predicate $pred_q \in cond_q$ that is contained by $pred_e$. If condition $cond_q$ also contains $cond_e$ both queries are equal. The filter condition is $cond_{filter} = cond_q \setminus cond_e$.

Example 7.5 *Assume* $cond_e = \{0/name \sim= "holbein", \text{exists}(0/birthplace)\}$ *and* $cond_q = \{0/name \sim= "holbein", 0/birthplace \sim= "Augsburg", 1/title \sim= "scene"\}$*. In this case,* $cond_e$ *contains* $cond_q$ *because all predicates in* $cond_e$ *contain another predicate in* $cond_q$*. The numbers in the predicates (0,1) refer to the position of a concept set in the corresponding derived concept sets. For example,* $\text{exists}(0/birthplace)$ *means the birthplace value has to be existent for an artist object. This means that if* $0/birthplace \sim= "Augsburg"$ *is true, i.e., a painter from Augsburg, the exists predicate will be true, too. The filter condition is* $\{0/birthplace \sim= "Augsburg", 1/title \sim= "scene"\}$*.*

In summary, a cache entry e contains a query q, i.e., $e \supseteq q$ if $dc_e \supseteq dc_q$ and $cond_e \supseteq cond_q$. The cache part of the result is constrained by the condition $cond = cond_{dc} \wedge cond_{filter}$. The cache entry matches the query, i.e., $e \equiv q$, if $dc_e \equiv dc_q$ and $cond_e \equiv cond_q$. The filter condition is empty in this case.

Cache Lookup Method

The **lookup** method returns a set of cache entries and a filter query to answer the actual query. Thereby, the filter query must not contain a keyword predicate. This constraint exists because keyword queries require specific keyword indexes, which the cache does not provide. However, concept and property existence selections are possible. The cache lookup handles two problems. For a filter operation, the lookup

method provides a set of cache entries that contain the query result and support together the complete keyword query. In the case of a semi-join result, the lookup method provides the best containing entry. The system filters the result by computing the join with the left input of the semi-join. Finally, the lookup method provides a filter in both cases using the definitions above. Algorithm 7.3 outlines the lookup method. In the remaining section, we will discuss the used method in detail. As the running example, we use the cache content depicted in Figure 7.10 on page 160 and the partial query plan $a' = \sigma_{name \sim= "holbein" \ and \ birthplace \sim= "augsburg"} \mathbf{ext}(Painter)$ and $g' = a' \ltimes_{JM(createdBy)} \mathbf{ext}(Graphics)$. Both expressions are translated into the corresponding plan operators.

Algorithm 7.3 Cache Lookup

```
    Input: plan operator p
    Output: E_q cache entries satisfying q and filter query cond_filter
 1: function CacheLookUp(p)
 2:     /* dc_p is derived concept set, cond_p selection condition */
 3:     (dc_p, cond_p) := createQueryRep(p)
 4:     cond_filter := ∅
 5:     if type(p) = FilterPOP then
 6:         E_cand := getAllCandidateEntries(dc_p, cond_p)
 7:         E_p := getBestSubset(E_cand, cond_p)
 8:         cond_filter := cond_p \ ⋃_{e∈E_p} e.cond
 9:         cond_filter := cond_filter ∪ getConceptSelections(E_p)
10:     else if type(p) = SemiJoin then /* Semi-join POP */
11:         e_best := getBestEntry(dc_p, cond_p)
12:         cond_filter := getConceptSelections(E_p)
13:         if e_best ≠ null then
14:             E_p := {e_best}
15:         end if
16:     end if
17:     if E_p = ∅ then
18:         return (∅, ∅) /* No Cache Entry found */
19:     end if
20:     return (E_p, cond_filter)
21: end function
```

The input of Algorithm 7.3 is a plan operator p. The plan operator p is translated into a query representation. The query representation consists of the derived concept set dc_p and the complete condition $cond_p$ using function **createQueryRep** (line 3). For example, for a' the values are $dc_{a'} = Painter$ and $cond_{a'} = \{name \sim= "holbein" \ and \ birthplace \sim= "augsburg"\}$. For the semi-join q', the derived concept set is $Painter/createdBy[name = "Graphics"]$ and $cond_{q'} = cond_{a'}$.

Now, we distinguish between query filter operation (lines 5-9) and semi-join operation (line 10-15). In the first case, the algorithm receives all cache entries E_{cand} that contain the query result. E_{cand} contains candidates for the best cache entry set E_q. The function **getAllCandidateEntries** uses two steps:

1. search for matching cache entry lists using dc_p, and subsequently,

2. search for cache entries using condition $cond_p$.

We test all concepts in the cache if they are equal to the final concept or super-concept of dc_p. For example, for $dc_{a'}$ the system looks into the concept $Painter$ but also $Artist$ or $Person$. All derived concept sets dc_e are selected that contain $dc_{a'}$. In this case, the set C_0 must include the concept $Painter$. For each of the selected concept sets, we add cache entries e to E_{cand} with $q(e) \supseteq q(p)$. This means, $e.cond$ contains $cond_p$. For example, cache entries e_4 and e_6 are selected for a'.

The function **getBestSubset** uses the candidate set to compute the best cache entry set containing the query result and comprising all keyword selection predicates of $cond_p$. We define the best set as the set with the smallest size, i.e., $\min \sum_{e \in E_q} e.size$. To find this set, we sort all entries in E_{cand} in increasing size order. The algorithm takes the first entry from E_{cand} and adds it to E_q. If all keyword containment predicates are satisfied, the algorithm will return E_q. Otherwise, the function takes the next entry in E_{cand} and adds it to E_q if it provides at least one further keyword containment predicate. In that way, function **getBestSubset** creates the set E_q. If E_q does not cover all keyword containment predicates after using all candidates, the function returns an empty set. In the running example, the set E_q consists of the cache entries e_4 and e_6 because the union of their conditions equals $cond_{a'}$. The filter condition is empty. Also, we do not have to add any concept selections because the intersection of e_4 and e_6 will deliver the result for a'.

For a semi-join, e.g., the computation of g', we use the following approach. We assume g' will be joined with a' in a later stage to $a' \bowtie g' = a' \bowtie (a' \rtimes_{JM(createdBy)} \mathbf{ext}(Graphics))$. Given an object set $a^* \supseteq a'$, it holds $a' \bowtie (a' \rtimes_{JM(createdBy)} \mathbf{ext}(Graphics)) = a' \bowtie (a^* \rtimes_{JM(createdBy)} \mathbf{ext}(Graphics))$. a^* filters all necessary objects from the extension of graphics, but leaves additional objects in the result. These additional objects are removed by the join with a'. Therefore, we try to find a cache entry that contains $dc_{g'}$ and $cond_{g'}$ using the function **getBestEntry**. The function returns the smallest of these cache entries. In the example, it is cache entry e_3. However, cache entry e_3 contains also objects of the sub-concepts that are not necessary for answering the query. Therefore, we add the predicate $concept \in \{Graphics\}$ to the condition $cond_{filter}$ to select only graphics objects.

Algorithm 7.3 returns an empty result, if E_p has been empty (line 15). Otherwise, it returns the cache entry set E_q and the filter condition $cond_{filter}$. In the running example, the algorithm returns $E_{a'} = \{e_4, e_6\}$ and $cond_{a'} = \emptyset$ for the filter query a' and $E_{g'} = \{e_3\}$ and $cond_{g'} = \{concept \in \{Graphics\}\}$ for the semi-join between a' and "graphics". In summary, the query $a' \bowtie g'$ is rewritten using the cache into

$$\left(\bigcap_{e \in E_{a'}} e.data \right) \bowtie \left(\sigma_{concept \in \{Graphics\}} e_3.data \right).$$

7.3.2. Cache Management

During query processing, the system adds intermediate results to the cache using the cache method **addEntry** . The function **addEntry** uses as inputs the derived concept set *dc*, the selection condition *cond* as well as the data set *data*. It searches for the derived concept *dc* in the first step. If *dc* has not been found, the cache creates a new derived concept set entry *dc* and a corresponding empty cache entry list. After having found or created a cache entry list, the system adds a new entry to the list. The cache entry contains all necessary meta data like the size of the data set, creation timestamp, and type of the data.

In this version, the system does not use semantic regions as proposed by Dar et al. [DFJ+96] and Karnstedt [Kar03]. Semantic regions are especially a tool for adding the results of complementary queries, which we do not use. However, we can support semantic regions in the following way. As described above, the cache allows as filter predicates category and concept selection as well as value existence test. Keyword containment queries are not considered. Thus, we define regions by combinations of keyword containment predicates. That means that we add all objects of a derived concept and a keyword containment condition to one set. Entries in a region have all the same keyword containment predicates and possibly additional category, concept, and exists predicates.

If adding a new entry exceeds the maximum cache size, the system will have to evict a set of entries. There are different strategies possible. Every entry has a certain *benefit*. The benefit depends on classical cache information like number of accesses or last access as used in LRU or LFU approaches [SSH05]. However, in distributed, heterogeneous systems the costs of creation of a result are variable. It is necessary to cache and materialize results that are expensive [AKS99]. In particular, bind join results are hard to compute in our scenario. Therefore, we combine the classical cache decision criteria with the cost criterion to a benefit value. A possible benefit calculation is presented in Equation (7.10):

$$benefit(e) = e.cost \cdot e.hits. \tag{7.10}$$

Here, the benefit of a cache entry e depends on its costs and the number of hits. To evict entries for storing a new entry, we sort all current entries in ascending benefit order. Now, the eviction algorithm removes entries from the top of this list until the new entry fits into the cache.

The benefit of a cache entry decreases over time if the access patterns change. One solution is the “aging” of statistics [SSV96]. We implement the aging in the following way. Every time when the cache reaches the maximum, we reduce all benefit values of cache entries by a certain factor. For example, we reduce the number of hits. That means that the benefit of older entries will be reduced over time.

We do not monitor changes in the local data. That means that we do not update cache entries. Instead, we invalidate cache entries after a specified timespan $maxKeepTime$. The old cache entries are evicted to reflect possible changes in the underlying sources.

7.3.3. Single Query Cache Use

We discuss the use of the cache for single queries. During optimization of a materialization, we use the cache to replace filter and semi-join queries with cache accesses. In this way, we implicitly reuse the results of previously executed queries during keyword processing. For this task, we modify the optimization algorithm 7.2 to use semi-joins and the cache. In every step, in which the algorithm creates a filter or semi-join plan operator, the optimizer tests whether cache entries exists to provide the answer. In this case, a cache retrieval operator is added to the plan. The adapted optimization approach is sketched in Algorithm 7.4. In the initialization phase, every sub-plan consisting of ConceptPOP, FilterPOP, and MaterializePOP is tested, if the result can be obtained from the cache. If this is the case, it is replaced by a CachePOP.

In the second phase, sub-plans are connected to joins step by step. If the new operator is a semi-join, the algorithm checks the cache, if the data is already materialized. In this case, the semi-join is replaced by a CachePOP.

The optimization ends with the selection of the best plan p from the plan set $plans(Q)$. The function **finalize**(Q) adds additional conditions and makes the plan ready for execution.

Example 7.6 *For illustration, consider the cache in Figure 7.10 and the query outlined in Figure 7.12(a). In the first phase, all three queries are added as a single plan operators. Thereby,* q_1 *is replaced by the cache entries* e_4 *and* e_6*. During the second phase, it is found* $q_1 \rtimes q_2$ *is in the cache (cache entry* e_3*.) The corresponding filter condition is* $cond_E = \{concept \in \{Graphics\}\}$*. The best final plan is as depicted in Figure 7.12(b).*

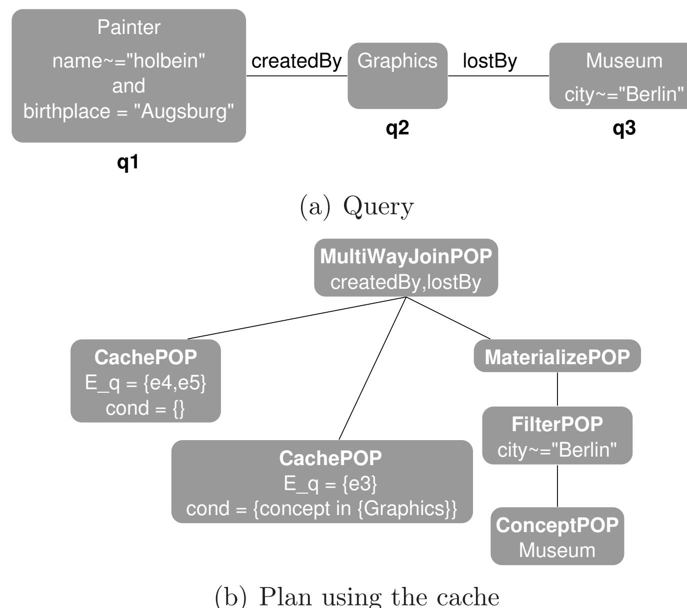


Figure 7.12.: Example plan for cache usage

Algorithm 7.4 Adapted Dynamic Programming Optimization

```
Input: query network (Q, E)
       cache – semantic cache
Output: best plan p
 1: function OPTIMIZE((Q, E), cache)
 2:   /* a set of pairs: (Q', P) with Q' ⊆ Q and P a list of plan operators */
 3:   plans := ∅
 4:   /* initialize */
 5:   for all q ∈ Q do
 6:     p = createPlan(q) /* Root is either a MaterializePOP or ConceptPOP or
      an unspecified FilterPOP */
 7:     if q is specified then
 8:       (E_q, cond_E) := cache.lookup(p)
 9:       if E_q ≠ ∅ then /* Found an entry set */
10:         p = createCachePOP(E_q, cond_E) /* override original plan */
11:       end if
12:       add ({q}, p) to plans
13:     end if
14:   end for
15:   /* start dynamic programming */
16:   for 1 ≤ i < |Q| do
17:     for all Q' ⊂ Q with |Q'| = i do
18:       /* Q' is valid, if it contains a specified query */
19:       if plans(Q') ≠ ∅ and Q' is valid then
20:         for all (Q'', P) ∈ plans : Q'' ∩ Q' = ∅ and Q'' and Q' connect do
21:           P' := createPlans(plans(Q'), P)
22:           for all p ∈ P' do
23:             if newOperation(p) is of type "SemiBindJoinPOP" then
24:               (E_q, cond_E) := cache.lookup(newOperation(p))
25:               if E_q ≠ ∅ then /* Found an entry set */
26:                 replaceSemiJoin(E_q, cond_E, p, newOperation(p))
27:               end if
28:             end if
29:           end for
30:           plans(Q' ∪ Q'') := plans(Q' ∪ Q'') ∪ P'
31:         end for
32:       end if
33:     end for
34:     prunePlans(plans, Q, i + 1)
35:   end for
36:   let p ∈ plans(Q) the plan with minimal cost
37:   finalize(p)
38: end function
```

7.4. Optimizing Query Result Reuse

The single materialized query approach separately optimizes queries and also decides locally, which intermediate results to cache or to use. However, keyword queries generate large sets of materialization queries, which have a large overlap. It is possible to reduce query costs further by using statistics from the generated query set. We use the statistics to "manipulate" the optimization in order to improve the materialized data use.

We will discuss three approaches in our query processing model:

1. the multi-query optimization of a query list network. A query list network represents a set of similar queries. In particular, if the system processes a complete query list network, then it will have to execute many overlapping queries. Thus, a multi-query optimization of the query list network promises a better query performance by smaller caches, fewer source queries, and fewer transported objects.

2. the second approach combines multiple query list networks. The optimization of a query list network considers already optimized and executed query list networks and tries to minimize the costs based on that information. Now, we also consider the optimizations of previously executed query list networks and build a shared query plan.

3. in order to use the cache more efficiently, we use query set statistics to estimate the benefit of materialized results within a keyword query. In that way, we calculate the benefit of a cache entry for a complete keyword query.

7.4.1. Query List Network Optimization

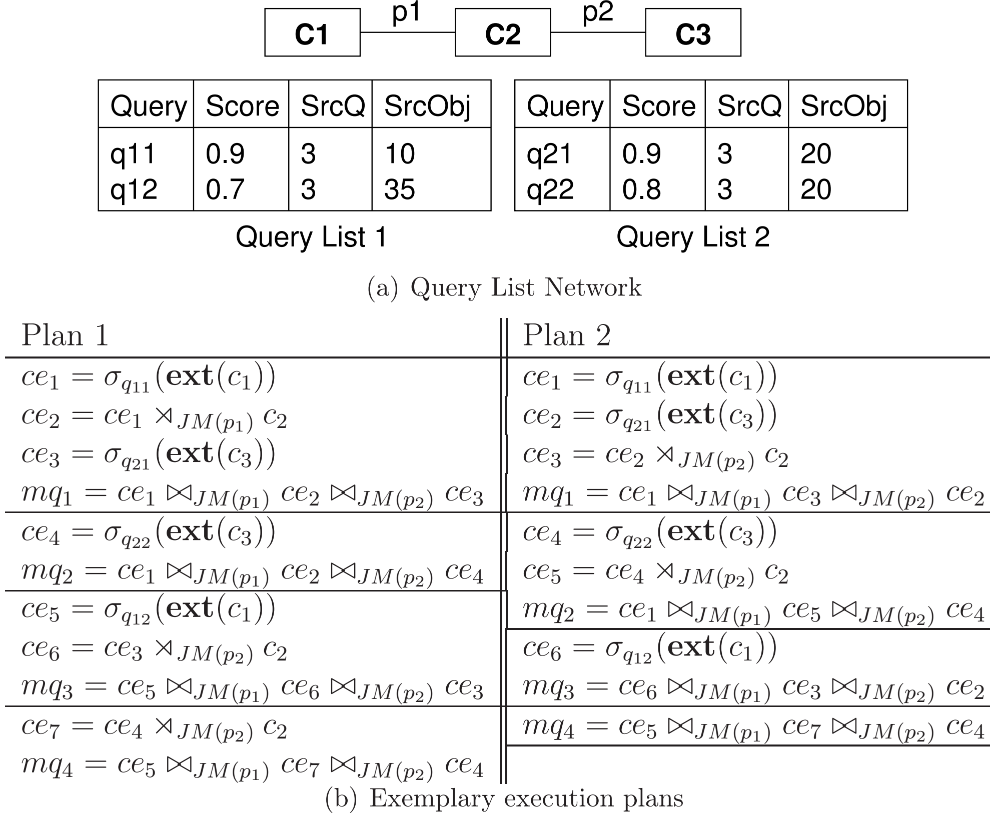
In a query list network, all queries of a query list network have the same structure but different conditions. We create one plan for all queries in order to minimize the average costs and to maximize the reuse of materialized results within one query list network. First, we discuss the complete processing of a query list network. Second, we present a motivation for top-k processing.

Complete processing

We motivate the approach using the example illustrated in Figure 7.13. Figure 7.13(a) shows a query list network consisting of two query lists and one free concept. The query list network represents four join queries. Assuming the semi-join approach as defined above, Figures 7.13(b) illustrates two possible query execution plans. Thereby, the results of materialization queries are materialized in cache entries ce_i. The join queries combine the intermediate result to finale object tuples.

Plan 1 assumes the use of the semantic cache and optimizes every query separately. This plan is not optimal with respect to the complete query list network. Plan 2 shows a better performance. It causes fewer cache entries and less source queries (40 instead of 50). In contrast to the first plan, the first query mq_1 is not locally optimal. Thus, it

is possible to improve the reuse of intermediate results within one query list network that includes a set of structurally similar queries.



Query	Score	SrcQ	SrcObj
q11	0.9	3	10
q12	0.7	3	35

Query	Score	SrcQ	SrcObj
q21	0.9	3	20
q22	0.8	3	20

(a) Query List Network

Plan 1	Plan 2
$ce_1 = \sigma_{q_{11}}(\mathbf{ext}(c_1))$	$ce_1 = \sigma_{q_{11}}(\mathbf{ext}(c_1))$
$ce_2 = ce_1 \rtimes_{JM(p_1)} c_2$	$ce_2 = \sigma_{q_{21}}(\mathbf{ext}(c_3))$
$ce_3 = \sigma_{q_{21}}(\mathbf{ext}(c_3))$	$ce_3 = ce_2 \rtimes_{JM(p_2)} c_2$
$mq_1 = ce_1 \bowtie_{JM(p_1)} ce_2 \bowtie_{JM(p_2)} ce_3$	$mq_1 = ce_1 \bowtie_{JM(p_1)} ce_3 \bowtie_{JM(p_2)} ce_2$
$ce_4 = \sigma_{q_{22}}(\mathbf{ext}(c_3))$	$ce_4 = \sigma_{q_{22}}(\mathbf{ext}(c_3))$
$mq_2 = ce_1 \bowtie_{JM(p_1)} ce_2 \bowtie_{JM(p_2)} ce_4$	$ce_5 = ce_4 \rtimes_{JM(p_2)} c_2$
$ce_5 = \sigma_{q_{12}}(\mathbf{ext}(c_1))$	$mq_2 = ce_1 \bowtie_{JM(p_1)} ce_5 \bowtie_{JM(p_2)} ce_4$
$ce_6 = ce_3 \rtimes_{JM(p_2)} c_2$	$ce_6 = \sigma_{q_{12}}(\mathbf{ext}(c_1))$
$mq_3 = ce_5 \bowtie_{JM(p_1)} ce_6 \bowtie_{JM(p_2)} ce_3$	$mq_3 = ce_6 \bowtie_{JM(p_1)} ce_3 \bowtie_{JM(p_2)} ce_2$
$ce_7 = ce_4 \rtimes_{JM(p_2)} c_2$	$mq_4 = ce_5 \bowtie_{JM(p_1)} ce_7 \bowtie_{JM(p_2)} ce_4$
$mq_4 = ce_5 \bowtie_{JM(p_1)} ce_7 \bowtie_{JM(p_2)} ce_4$	

(b) Exemplary execution plans

Figure 7.13.: Example query processing

The example shows, the reuse can be maximized if firstly, the query plans have the same structure, i.e., always use the same query list for a semi-join. Secondly, the best of these structures has been selected. For this, the used approach summarizes the costs of every query list. The best structure is found by using summarized costs and the optimization algorithm 7.2. The result is a query plan template that is used for every query generated by the query list network.

The cost values are estimated as follows. The query list statistics consists of the number of source queries ($estSrcQ$) and the number of transferred objects ($estSrcQbj$). The latter is also the estimated size of all queries ($estSize$) in the query list. Let $QL_i = [q_{i1}, q_{i2}, \ldots, q_{in}]$ be the query list with n queries, the costs are computed as the sum of the single query costs:

$$estSrcQ(QL_i) = \sum_{1 \le j \le n} estSrcQ(q_{ij}) \tag{7.11}$$

$$estSrcObj(QL_i) = \sum_{1 \le j \le n} estSrcObj(q_{ij}) \tag{7.12}$$

$$estSize(QL_i) = estSrcObj(QL_i) \tag{7.13}$$

For optimization, we create one summary query network from the query list network. The specified queries represent the corresponding query lists. For example, the query $q_i^* = \bigcup_{1 \leq j \leq n} q_{ij}$ represents the query list QL_i. The costs of the query q_i^* are estimated by using defined cost functions. The optimizer uses this summary query network as input and produces the template plan. The template is used for every query in the query list network.

The cache and the empty result detection are used in the following way for every query. First, the plan is parameterized. Before executing the parametrized plan, the system tests if intermediate results can be retrieved from the cache. If this is possible, the bind-joins are rewritten to joins. Otherwise, the system adds materialize operations to the plan in order to cache the intermediate results.

Example 7.7 *Given is the query list network in Figure 7.13(a). If the optimizer used summarized costs, the optimization produces the plan template:*

1. $c_1' = \sigma_{cond}(\boldsymbol{ext}(c_1))$
2. $c_2' = c_1' \rtimes_{JM(p_1)} c_2$
3. $c_3' = \sigma_{cond}(\boldsymbol{ext}(c_3))$.
4. $c_1' \bowtie_{JM(p_1)} c_2' \bowtie_{JM(p_2)} c_3'$.

During execution, we replace the template for c_1' and c_2' by the corresponding queries q_{1i} and q_{2j}. In that way, we create plan 2.

Top-k processing

The described approach is appropriate for processing a complete query list network. During top-k processing, there is the possibility that the system does not to process the complete query list network, i.e., all queries in it. Therefore, we modify the approach. Every step induces a set of queries. We optimize this set at once taking previous results into account.

Example 7.8 *Consider the example illustrated in Figure 7.13(a). In the first step, the query mq_1 is executed. In the second step, the algorithm creates and executes query mq_2. The third step, query q_{12}, induces two queries mq_3 and mq_4 that have to be optimized together to enable the best performance.*

For every step, we know that all previous steps have been executed. It follows that we have to consider only the connection between the currently selected query and the already executed queries. For example, if we select query q_{12}, queries q_{21} and q_{22} were already executed. Furthermore, the joins are processed. It follows that we have only to consider the connection between q_{12} and the query list of c_3. In particular, the system has to decide, which bind joins have to be executed.

Let c_k be the currently selected concept and q_{kj} the selected query of the query list QL_k. The concepts c_{k_i} are the concepts that are adjacent to c_k, either directly or via free concepts. The query list $QL_{k_i}^m$ contains all already executed queries of QL_{k_i}. We

optimize this part of the query list network. All other joins are not necessary in this phase, because their results are already materialized.

Consider the join between the selected query q_{kj} and the query list network QL_i^m and a free concept c. We have to create a query plan template for all queries created in this step. The approach is equivalent to the complete computation. We summarize all queries in QL_i^m to one query template. Then, we create a query network and optimize using the presented dynamic programming algorithm. For this, we adapt the costs as follows. Equations (7.14) and (7.15) summarize the cost of a bind join between a query list QL_i^m and the free concept c. The cost function tests for each query, if the result resides in the cache. The query list of the selected concept consists only of query q_{kj}.

$$srcQ(QL_i^m \bowtie c) = \sum_{1 \leq j \leq n} \begin{cases} srcQ(q_{ij} \bowtie c) & \text{if } q_{ij} \bowtie c \text{ not in cache} \\ 0 & \text{otherwise} \end{cases} \tag{7.14}$$

$$srcObj(QL_k^m \bowtie c) = \sum_{1 \leq j \leq n} \begin{cases} srcObj(q_{kj} \bowtie c) & \text{if } q_{ij} \bowtie c \text{ not in cache} \\ 0 & \text{otherwise} \end{cases} \tag{7.15}$$

We use this cost estimations as input for the dynamic programming algorithm and create a first plan.

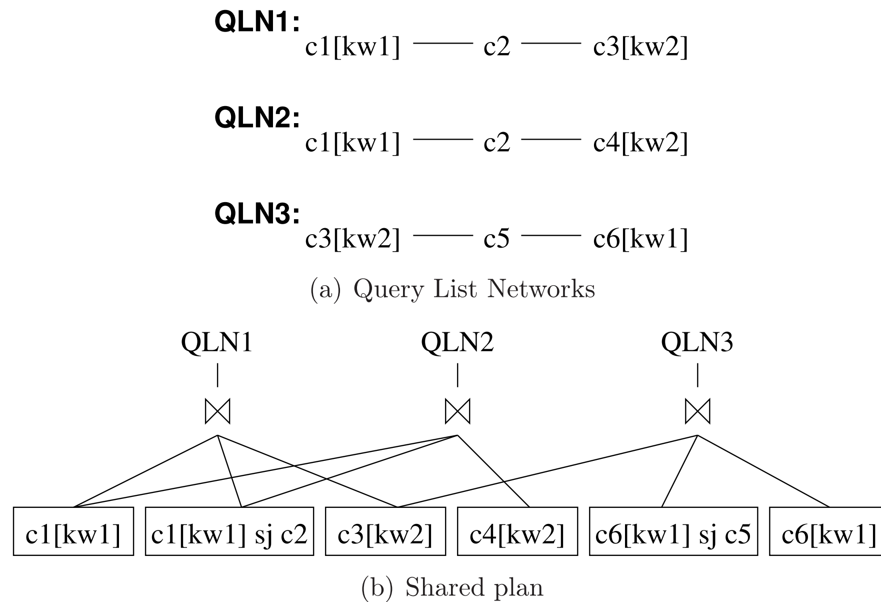
Example 7.9 *Continuing the previous example, we select the query q_{12} that is connected to query list 2 via concept c_2. QL_2^m contains both query q_{21} and q_{22}. As both queries of QL_2^m and their representing joins have been executed and cached by the system, the join between QL_2^m and c_2 does not induce source queries. Hence, the optimizer returns the query plan $q_{12} \bowtie_{JM(p_1)} (QL_2^m \rtimes_{JM(p_2)} c_2) \bowtie_{JM(p_2)} QL_2^m$ as template.*

7.4.2. Query List Network-based Query Reusing

The previous section described the optimization of a query list network. Thereby, the optimization handles every QLN separately. In this section, we share intermediate results between query list networks to improve the reuse.

Figure 7.14 illustrates the proposed approach. Figure 7.14(a) shows three query list networks that are processed in descending order of their maximum possible score. In the first step, the query list network QLN_1 is optimized using the approach in the previous section. It uses the plan for QLN_1 in Figure 7.14(b). In the next step, the query list network QLN_2 is selected. During the optimization of QLN_2,, the optimizer checks, if it shares parts with previously used query list networks. Therefore, two semi-joins of QLN_1 are reused. Finally, QLN_3 reuses one semi-join. In this way, a common plan is built step-wise in top-k order.

We now describe the approach in detail. We define the data structure to hold the shared plan. Based on this, we describe the optimized construction of the shared plan.



(a) Query List Networks

(b) Shared plan

Figure 7.14.: Shared query list networks

Data structure

The shared plan consists at the first level of a set of filter and semi-join query templates. We denote the templates *materialization nodes*. Every materialization node represents a list of single concept queries or bind semi-join queries. In consequence, a materialization node represents either a query list or a bind-join between a materialization node and free concept. Thus, we use a similar description as for cache entries. Every materialization node is identified by a derived concept set *dcs* and the keyword set KW. The former identifies the derived concepts that are used in the materialization node. Every node represents a set of queries. Every query uses a selection predicates. The keyword set KW contains the keywords that are used in the selection predicates.

At the second level, we have a set of join nodes. Every join node represents one query list network. Every join node uses a subset of materialization nodes, to complete the query plan template.

Construction of a shared plan

The shared template plan is constructed during the processing of query list networks. We assume that we use the spares algorithm proposed by Hristidis et al. [HGP03] in the execution. That means that the query list networks are handled in descending score order. We add new materialization nodes as they occur.

If the system encounters a query list network that has not been used before, it will optimize it using the approach in Section 7.4.1. During optimization, the algorithm checks if a materialization plan operator is already in the shared plan. If this is the

case, we will modify the costs of computing the results by setting the number of source queries and source objects to zero. That means:

$$srcObjects'(p) = \begin{cases} 0 & if\ p \in plan \\ srcObjects(p) & \text{otherwise} \end{cases} \quad (7.16)$$

$$srcQueries'(p) = \begin{cases} 0 & if\ p \in plan \\ srcQueries(p) & \text{otherwise} \end{cases} \quad (7.17)$$

with p a materialization plan operator and $plan$ the current shared plan. The system optimizes the query list network by using the modified costs. The result is a plan consisting of materialization plan operators and one join operator as root. All new materialization plan operators are added to the shared plan at first level. The join operator as join node of the query list network is added to the second level. The modified cost model ensures that the biggest overlapping is found and selected during construction.

Adapted cost model

The cost model is only a binary model, i.e., if a materialization node is in the plan, the cost values are zero. In some cases, different plans are used with different costs and different benefits for the complete keyword query. In this case, we use the following approach.

The benefit of the materialization depends on the cost of the materialization and the number of usages in the keyword query. We count the number of possible usages of a semi-join or filter operation during the query list network enumeration. Having this number and the cost estimations for the plans of a query list network, we create a cost model for the benefit of an operator $benefit(op)$. The benefit is high, if the computation costs is low and the number usages are high. We select than the plan of the query list network with the highest benefits. The benefit of an operator is computed as

$$benefit(op) = \frac{nrUsages(op)}{cost(op)}. \quad (7.18)$$

The term $nrUsages(op)$ denotes the number of usages of the result of op and $cost(op)$ the costs of computing the result. Note, the operator op is a template for the materialization queries.

We use the previously defined optimization algorithm. Every query list network is optimized when it is used the first time. By using the benefit cost model, we can find the globally most beneficial plan for this query list network with respect to the previously executed query list networks.

7.5. Query List Optimizations

Compact concept graphs significantly improve the performance of the query list network enumeration. However, the single concept query lists are larger than in the

original concept graph because of the combination of the lists of the sub-concepts into one list. A keyword query with many keywords, which occur in many properties of one concept, also creates a large query list. In this section, we propose two optimizations for these cases. First, we split query lists, i.e., we execute sub-queries if they are used in other query list networks. Second, single concept queries in a compact query list node that have the same condition are merged into one query that includes all concepts and the condition.

7.5.1. Splitting Query Lists

Assume a long keyword query and a concept with a high number of properties. If the keywords occur in many properties, the algorithm will create a large number of single concept queries. However, it is highly possible that many of these queries are empty. The given empty result detection algorithm in Section 7.2 cannot efficiently handle this case because covering queries are not previously executed. One solution is to generate and execute all covering queries and combine their results to the final results.

Assume a query list c^{KW} for the concept c. Every query $q \in c^{KW}$ has a condition $cond_q$ with $|KW|$ predicates. We denote q as $q = c[cond_q]$. Furthermore, we assume that query lists $c^{KW'}$ with $KW' \subset KW$ are also used in query list networks. Based on these assumptions, we execute every query $q \in c\{KW\}$ as follows. First, we execute every predicate pair of $cond_q$, i.e., execute the queries $c[pred_i \wedge pred_j]$ for all $pred_i, pred_j \in cond_q$ with $i \neq j$. Subsequently, we create query results with three predicates by using the already materialized results. In the following steps, we create the results of all conditions sizes up to $|KW|$. If any of the query $c[cond']$ is empty, we add the query to the empty statistics and return an empty result. In that way, the execution is more expensive in the first place, but stores intermediate results for future queries and allows more efficient empty result detection. Thus, it helps for large query lists and a higher number of k in top-k queries.

7.5.2. Merging Single Concept Queries

The second type of optimization of long query lists is the merging of single concept queries of compact concept nodes. In a compact query list, queries use different concepts of one concept hierarchy. For example, we have a compact node `culture assets/*` containing all concepts that describe cultural assets and a compact query list `culture assets/*{flowery, gogh}`. The compact query list may contain the single concept queries `fine art[title∼="flowers" and artist∼="gogh"]` and `paintings[title∼="flowers" and artist∼="gogh"]`. Both queries have the same selection condition. Thus, it is possible to combine the queries to `(fine art union paintings)[title∼="flowers" and artist∼="gogh"]`. That means that the mediator executes both queries at once. This approach enables the application of the optimization of Yacob [SGS05]. For every source, the optimizer checks if a concept contains sub-concepts, and only the super-concepts are executed by the mediator.

We merge queries as follows. Initially, we add two numbers to every single concept query. The *groupid* identifies the group of the query, and second, the *groupsize* the

number of queries in the group. We assume the predicates in the conditions are ordered by the keyword. We scan the query list twice. In the first scan, we add a *groupid* to the queries and collect the counts in a hash table. In the second scan, we add the query sizes to the single concept queries. Single concept queries belong to the same group, i.e., they have the same *groupid* if they have the selection condition.

During the execution of a query list, we use the following approach. When we execute a query that is not in the cache, we collect all queries with the same query group *groupid*. We use the parameter *groupsize* to stop the search of queries of the group. We will stop the scan, if the number of found queries equals *groupsize*. Now, we union the concepts and execute the merged query. The results are added to the cache for each concept separately. The results are reused in later materialization queries.

The advantage of the approach is the possibility of optimization and reduction of mediator calls. However, the approach also induces an overhead for query grouping. Furthermore, for the first queries of every query group, the costs are higher than without merging because we execute later queries in the same moment. This is disadvantageous in top-k queries.

7.6. Discussion

The contributions of this chapter are in the fields of empty result detection, semantic caches, and plan optimizations for keyword queries.

Empty result detection

The first part deals with empty result detection. Empty result detection is used in relational databases, e.g., [Luo06]. Here, empty sub-plans are efficiently detected after optimization. Thereby, the physical plans are transformed back into logical plans. An efficient data structure based on first, relation name sets and second, predicate checking is used to detect empty sub-plans. In contrast, we execute the empty result detection during optimization. We extend the data structure for concept-based queries. The approach of Demidova et al. [DZN10, DZZN09, ZZDN08] directly use information of empty partial queries for keyword queries. The authors test every partial materialization query if they are empty. However, this approach is expensive in a distributed environment with limited access patterns. Thus, empty result detection must be combined with the execution of the queries. We utilize this approach only for completely materialized queries.

Semantic cache

The semantic cache is the second optimization to reuse intermediate results. There is a large body of research about semantic caches and in particular, semantic caches for mediator systems or other integrating systems, e.g., [Sel88a, DFJ+96, KB96, ACPS96, LC99, LC01, CRS99, AKS99, Kar03, KSGH03, WAJ08, WA10]. Semantic caches and materializations are crucial in mediator systems to mitigate the disadvantage of virtual integration. All studies show that using a semantic cache or selective materializations

improve the performance compared to tuple or page level caches known from centralized database systems.

Dar et al. [DFJ+96] introduce a semantic cache based on semantic regions. A semantic region represents a set of tuples that is described by constraint queries. Regions are merged based on heuristics that balance between supporting efficient cache eviction and supporting large query results. Dar et al. also describe fundamental query processing with a semantic cache that we use in this thesis. Keller and Basu [KB96] present a cache schema based on predicates. Their main focus is the update of the client caches. Höpfner describes the efficient notification and update of a large number of client caches using a Trie-based data structure [Höp05]. Another optimization of containment checking for many cache entries is based on templates [APTP03]. The authors propose the removal of common predicates from query statements to reduce the checking cost for large sets of queries. In our work, we do not consider updates on the client because the local sources are not cooperative and do not notify about changes. Instead, the cache content is invalidated either after one keyword query or a pre-configured time.

Lee and Chu describe a semantic cache system for Web sources [LC99, LC01]. They classify different levels where a cache can be used: in the mediator or the wrapper level. They use the wrapper level caches that describe queries using the local query model. The cache is a key-value table. The key is a conjunctive predicate and assumes that every source consists of one relation. The approach uses overlapping cache entries. The authors use a reference counter for data tuples to reduce the storage overhead. Because of overlapping cache entries, Lee and Chu argue that the best containing or contained match has to be found. They show a lattice structure that optimizes this task. Using additional information and constraints, more cache hits are generated. The approach in this work also uses overlapping cache entries, but finds the best set of entries to support keyword queries using a greedy algorithm. Cache hit rates are increases by using the relationships between concepts in the concept schema. We use the concept schema as external information. Adali et al. provide a semantic cache for the HERMES mediator system [ACPS96]. They also propose external knowledge to improve the cache hit rate. They use rules (called invariants) to replace parts of the query. The rewritten query is used as probe query in the cache.

Chidlovskii et al. propose a semantic cache for querying heterogeneous Web sources used by a meta-search engine [CRS99]. The meta-search engine uses conjunctive keyword queries with terms of the form `attribute op value`, where `attribute` describes an attribute like "title" or "body", `op` is either contains or equal, and `value` is a phrase. The approach also assumes that sources allow negation. Based on this query language, semantic regions (cache entries) are identified by conjunctive keyword queries. The cache is on mediator level, but the cache is organized by sources. The results are semi-structured. Besides traditional operational cases (equal, contained, and containing matches) between a probe query and a cached query, the approach also supports the one-term difference case. Here, all regions that have all but one keyword in their key are selected, and the query's part of the union of the entries and remainder query result are returned as results. In order to support heterogeneity, the approach modifies the cache results with respect to the completeness and checkability of the source. The cache replacement strategy is based on a weighted most-recently used value. Chidlovskii et

al. developed an approach for meta-searcher over documents and do not support structured data and join results. The one-term difference case is similar to our approach of using a set of cache entries to answer a query comprising keyword containment queries.

Ashish et al. describe the selective materialization data in a concept-based mediator system [AKS99]. It is based on the concept-based mediator SIMS(ARIADNE) (see Section 2.2.2). The selective materialization of data on mediator level follows the following three criteria: (*i*) user query distribution, (*ii*) cost of result generation with respect to limited query capabilities of Web sources, and (*iii*) update frequency of the sources. The materialized data is treated as one separate source. The materialized data is described by sub-classes of the domain model. The user queries are classified by their used concepts, predicates, and the required attributes for selection of the data. The number of classes is reduced by query class merging. The query classes are merged when they are siblings and have similar output attribute sets. Similar to our approach, the data is described by the concept model. However, the authors support single concept selections and not derived concepts. Derived concepts are necessary to support joins for keyword queries. Similar is the concept of avoiding expensive source queries. We replace bind-join operations by joins between two specified sources.

Karnstedt et al. propose a semantic cache as an extension of the YACOB mediator query processing system [Kar03, KSGH03]. The cache is located on mediator level. The cache is structured by the concept schema. For every concept, there is one list of cache entries. A semantic region describes a cache entry. A region is defined by a conjunctively connected set of predicates of the form `property op value`. The entries are disjoint. The approach supports all operational matches. Keyword containment predicates $prop \sim=$ "*value*" are supported using string inclusion. The approach proposed here uses parts of this concept, but extends it by using derived concepts.

Wang et al. proposed a query planning system for interdependent deep-web sources [WAJ08]. In order to improve the performance, Wang and Agrawal propose a mechanism to reuse previous results [WA10]. The system takes as input a set of selection predicates and output attributes. The system creates a query graph with goal to return all specified output attributes. It starts with the specified sources. The query plan is a directed graph of interdependent sources. Previous query results are cached and stored together with their query plans. The system finds subgraphs that can be answered by the cache. The operations resemble the bind-join in our system. Instead of graphs we use semi-join operation trees. Furthermore, we consider simpler query plans.

Lou and Naughton developed a cache for form-based keyword queries [LN01]. Form-based queries mean that the results of a query come from one relation and are ordered lists. The user queries are conjunctive keyword queries. Thus, the cache entries are ordered lists that are described by sets of keywords. They describe the matching classes of conjunctive keyword queries, ordered conjunctive keyword queries, and top-k ordered conjunctive keyword queries. The authors implement the system for conjunctive keyword queries. The study proposes one cache for each form. The cache consists of a number of conjunctive keyword queries that point to tuples in a common tuple set. Furthermore, the cache comprises a dictionary of keywords in the search attribute. That means, in contrast to our approach, it supports keyword containment filter queries. The difference to our work is that joins are not supported in this work.

Seltzsam et al. developed a semantic cache definition for Web services [SHK05]. The authors annotate WSDL definitions with cache relevant information like the order, which attributes are cache relevant, etc. The cache entries are described by web service requests. While we use a specific, optimized cache for keyword queries, Seltzsam et al. propose a general implementation for proxies for Web Services.

Materializations are also beneficial in centralized databases [Sel88a] in the form of materialized views [SSV96] or materializations during multi-query optimization [Sel88b]. An intriguing approach is proposed by Ivanova et al. [IKNG09, IKNG10]. In a column-oriented, main-memory database, intermediate results of query plans are materialized and reused in later queries. The intermediate results are identified by the plans that created them. Thus, by comparing new plans with cached results, one can rewrite the plan for re-using the cache content. This is similar to re-using semi-join results in the keyword search, but it has another application context and is used in a centralized database system.

Schema graph-based keyword search

The keyword search system in this thesis is a member of schema graph-based systems. Related are systems over relational databases and streams. In this thesis, we adapted the algorithms of Hristidis et al. [HGP03] and Sayyadian et al. [SLDG07] for the execution of query list networks. The algorithm exploits the monotonicity of the ranking function to evaluate candidate join queries. However, Hristidis et al. and Sayyadian et al. deal with tuple sets and not with query lists. While they focuses on reducing the number of joins in a top-k query, we focus on re-using as much intermediate query results as possible. Hristidis et al. and Sayyadian et al. execute every join separately without caching or combine a number of joins to a disjunctive query. We assume that only conjunctive queries are allowed by the source systems.

Hristidis et al. [HP02] propose an optimization algorithm for the evaluation of all candidate networks. It focuses on the evaluation of all results. It creates an order of joins in that way, that intermediate results are reused. The work is based tuple sets instead of query lists. Furthermore, it does not support top-k evaluation like the algorithms before.

Qin et al. [QYC09] also use semi-joins to filter free relations and reduce them to tuple sets that are joined. The reduced relations are joined in a second phase. In this way, they reduce the number and size of intermediate results if the candidate networks are entirely executed. We also exploit the concept schema and semantic cache ideas to allow better reuse of intermediate results. Furthermore, we allow optimization over different query list networks.

Related is also research about keyword search over relational tuple streams [MYP09, QYC11]. The problem is defined as follows: given are a conjunctive m-keyword query and a relational schema. Now, we require all minimal joining tuple networks that contain all keywords and that come from relational streams.

Markowetz et al. [MYP09] introduced the problem of keyword search over relational tuple streams. Their solution consists of candidate network generation and efficient network evaluation using an operator mesh. Every candidate network is translated into a left-deep operator tree consisting of joins and selections. Now, the operator trees are

merged into one mesh. The merging allows the sharing of join computations between candidate networks. This is similar to our shared plan. However, our plan is based on semi-join trees. The optimizer creates the operator trees to find the cheapest operator order. Qin et al. also use semi-joins instead of joins in the first phase [QYC11]. They improve the approach of Markowetz et al. by introducing the $\mathcal{L}$-lattice. The lattice is similar to merged operator trees, but the construction is based on candidate networks. A new network is merged in that way, that the overlap is biggest with the lattice. In the first phase, the tuples in every relation or stream window are reduced to those tuples that join to every adjacent relation. That means that it uses the semi-join approach. The reduction starts with the selection nodes. The work concentrates on obtaining all results. We also support top-k answers. Furthermore, our approach tries to find the most cost efficient overlapping.

Further schema-based keyword systems, e.g., [BRDN10, CBC+09, LLWZ07, LYMC06], deal with different optimization and ranking aspects. However, the optimizations do not deal with reuse of intermediate results. Data graph-based keyword search systems (see Section 4.2.4) do not apply here. The virtual integration requires the execution of queries created from a schema graph-based approach, because the global data graph is not materialized.

7.7. Summary

This chapter has the following main contributions. Initially, we described the query processing based on bind-joins and semi-joins. Using semi-joins, we can reduce the size of the cached and materialized intermediate results. We presented two basic execution algorithms for query list networks from the literature. They were optimized during the remainder of the chapter. The first optimization is the detection of empty results. The detection is based on query coverage. We defined query coverage for concepts and concept-based queries. The detection of empty results allows the skipping of empty result generating materialization queries. The second concept is intermediate query result reuse. Here, we defined query containment for concept-based queries. The first approach is a semantic cache structured by derived concepts. In order to improve the reuse rate, we optimized a query list network to one query plan template. Every materialization query had to use this template. Further improvement promises the generation of a shared plan. At last, query splitting and query merging were presented as further optimizations.

In summary, the main contribution is the description of the query list network evaluation based on empty result detection, semi-joins, and aggressive reuse of intermediate results to reduce the number of source queries and transferred objects. We argue that the presented optimizations allow the efficient schema graph-based execution of keyword queries over distributed, heterogeneous data sources. We validate the algorithms in the following Chapter 8 using a prototype.

8. Implementation and Evaluation

In this chapter, we present the developed prototypical implementation and experiments to validate the key aspects of the presented approaches. We describe the architecture and the implementation of the keyword search prototype in Section 8.1. In Section 8.2, we specify the evaluation goals and the used data and query sets. Section 8.3 investigates the efficiency of the keyword query processing. In Section 8.4, we evaluate keyword effectiveness. We conclude the chapter with a summary in Section 8.5.

8.1. Architecture and Implementation

The developed keyword search system is based on the Yacob mediator system described by Sattler et al. [SGHS03, SGS05]. The Yacob system is used for concept-based, integrated access to heterogeneous sources. We extend it by the keyword search component, which also comprises the join processing. The complete system is implemented using the Oracle Java SE 6[1]. The components use additional libraries and software products that we will describe in detail in this section. Figure 8.1 outlines the system architecture and the developed components. The keyword search components are highlighted and are the focus in this thesis. In the remaining section, we will describe the purpose and the implementation of every component.

Yacob mediator components

We sketch the function and the implementation of every Yacob mediator component. Details of the components are presented in [SGHS03, SGS05, KSGH03].

Concept management component. The concept management component manages the concept, categories, properties and the inter-concept properties as well as the mapping information. The complete integration schema is expressed in RDF. In order to manage the RDF data, the concept management component utilizes the Jena Semantics Web Framework[2]. The component realizes the concept schema access operations on a main memory RDF graph. The graph is loaded during startup from different RDF files.

Query planning component. The query planning component takes a CQuery statement as input and creates a query plan. In the first step, the planning component parses a CQuery statement and creates a tree of algebra operators. It rewrites the statement according to the rules presented in Sattler et al. [SGS05], which are also

[1] http://www.oracle.com/technetwork/java/index.html Last accessed: 2012-05-16
[2] http://jena.apache.org/ Last accessed: 2012-05-16

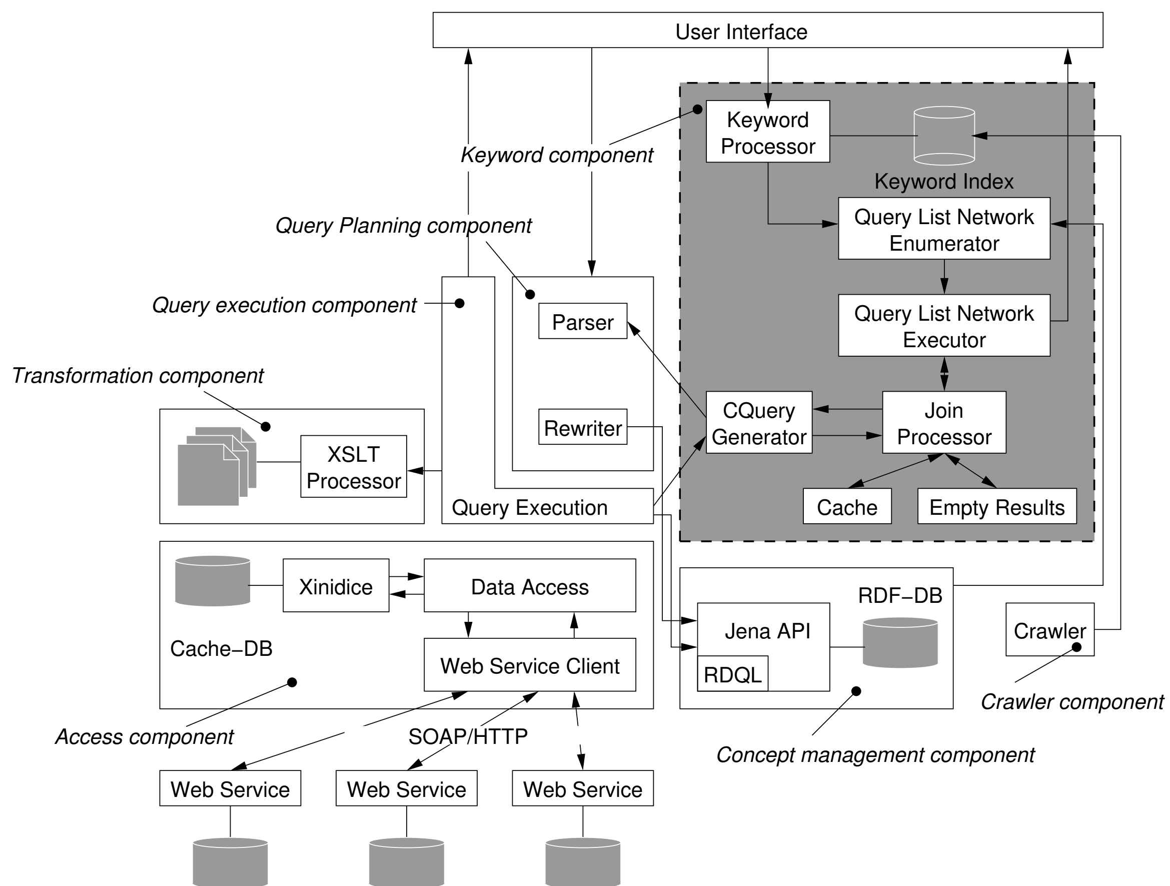


Figure 8.1.: System architecture

summarized in Section 3.3. The resulting query processing plan is sent to the query execution component.

Query execution component. The query execution component processes the query plan. It implements all necessary query operators for single concept query processing. In particular, the component integrates results using the extensional union operator. The source queries are executed in parallel threads. In summary, the query planning and the execution components implement the execution of single concept queries, but they do not support join processing.

Transformation component. The transformation component transforms local XML objects into global objects with respect to global concept schema. It consists of an XSLT processor, which applies created or provided XSLT transformation rules to local objects. Thereby element names are adapted. The component uses the standard XSLT processor provided by the Java development kit. The XSLT rules implement the mapping rules [SGS05] (see also Section 3.1.2).

Data access component. The data access component provides the communication to the sources. It receives an XPath query from the query execution component and

answers it from the included cache [Kar03, KSGH03] or sends the query to the local sources. The component invokes sources by Web service calls and assumes every source can answer a basic, conjunctive XPath query. The cache is not controlled by the keyword search component but is part of the mediator YACOB system. During the experiments, the cache is not used because we want to evaluate the join cache approach.

Crawler component. The crawler component obtains information from the sources in order to build source content descriptions and finally the keyword index. The crawler belongs to the keyword search components but is not in the focus of this work. Solutions are: *(i)* sources send descriptions during offline phases; *(ii)* the crawler samples the sources [Dec04]. The component stores the source content descriptions in a PostgreSQL[3] managed database. The keywords and their statistics are extracted using the full-text capabilities of PostgreSQL, which includes normalization of the terms.

Keyword search components

The YACOB components execute single concept queries. We now describe the keyword search and join processing components and their implementation.

Keyword index. The keyword index is implemented as one relation using PostgreSQL. Table 8.1 outlines the schema. It omits the category support. A full text index is used on the keyword column. Further indexes exist on the Dewey identifier, concept, and property columns. Dewey identifiers are implemented as bit strings. The index allows searching for Dewey identifier prefixes. The index is created from the source content descriptions and the integration schema index.

Field	Description
keyword	the keyword
keywordtype	the type of the keyword
dewey	the Dewey identifier of the concept
concept	the concept
property	the property
source_data	the sources with corresponding document frequencies
weight	the term weight value
sum_tf	the sum of all source term frequencies

Table 8.1.: Keyword index schema

Keyword processor. The keyword processor parses concept-based keyword queries and combines lists of index entries to single concept query lists. It is implemented in

[3]http://www.postgresql.org/ Last accessed: 2012-05-22

Java. For a plain query term, the processor creates one SQL query. For concept-based query terms, it executes three SQL queries and combines the index entries in main memory. We implemented concept-label query expansion with the help of Dewey identifier comparisons. The creation of single concept queries is executed in main memory for a set of keywords and index entry lists, respectively.

Query list network enumerator. The query list network enumerator uses the single concept query lists from the keyword processor as well as the concept schema graph from the concept management system. Initially, it creates the annotated concept graph in main memory. The single concept query lists are held in a Java implemented data structure. All compacting and enumeration algorithms are implemented in Java on the concept graph data structure.

Query list network executor. We implemented the different approaches of the query list network executor in Java. The executor creates join query plans. It uses the optimizer, the statistics, and the cache metadata to produce query plans. The query plan is sent to the join processor. The executor retrieves the answer from the join processor, adds metadata to it, and adds it into the result set. If the result is complete with respect to the keyword semantics, the executor will return the result to the user.

Join processor. The join processor implements the plan operators described in Section 7.1.1. The filter plan operator is implemented using the CQuery generator. The bind join operator uses the results of the mediator system to create new mediator queries and join the results. The materialize operator stores the input data set in a main memory database managed by the system H2 database[4]. The result is stored into a table containing the XML objects as well as separate columns with the join property values. A hash index is added to every join column to support fast global join processing. The multi-way join is implemented as join over all materialized tables. The CachePOP is implemented as follows. Given a set of cache entries a SQL view is created over the cache tables. The view implements the intersection of the best cache entries. The view is used by the global join and in bind join operations. The system retrieves relevant values from a join property column during a bind join. After all necessary data is materialized in H2 tables, the global join is executed as SQL query.

CQuery generator. The CQuery generator is based on the work of Declercq [Dec04] as well as on the works of Geist et al. [GDSS03, Gei04, SGS05]. It takes an index entry tuple and creates single concept query plans. However, we modified the generator in this study. Now, the CQuery generator creates actual query plans for the mediator system instead of CQuery statements. In that way, no parsing is necessary, but the mediator still applies query optimization rules.

Cache. The cache metadata is kept in main memory using a Java hash table. The data is stored in a H2 main memory database. We add join relevant attributes into separate columns and add indexes to it.

[4]http://www.h2database.com/ Last accessed: 2012-05-20

Statistics. The statistics are also held in main memory. Statistics comprise the sizes of the concept extensions for every source. Document frequencies of terms are stored in the index entries of the keyword index. Using these both statistics, we estimate the costs during join optimization. Finally, the statistics comprise the empty result statistics, too. Empty result statistics are held in the main memory to allow fast access during optimization.

8.2. Scenarios and Evaluation Measures

In this section, we describe the evaluation goals and the evaluation scenario. In the literature, there are two related approaches: keyword search systems over databases and mediator systems. Schema graph-based keyword search systems are comparable to the proposed approach. However, these systems require either a centralized database or complete query access to distributed databases, i.e., general joins. We assume that we only have limited access patterns. Hence, the approaches are hard to compare quantitatively. Because of bind join operations, many source queries are generated causing long query execution times. In contrast, Mediator systems do not support finding of connected objects that contain keywords. Several approaches support multi-media queries and merge ranked results, but they do not provide a keyword search over different objects. Hence, we investigate only the efficiency of the proposed optimizations.

The second problem is the effectiveness of keyword queries. We use a standard ranking function, which might not provide the best effectiveness. However, we want to validate if concept-based keyword queries allow better and faster query processing. Faster means that the user-provided labels allow the restriction of the number of possible candidate queries, and the query expansion allows retrieving relevant results. In summary, we validate the efficiency and the effectiveness of the system.

In the remainder of this section, first we discuss the evaluation goals and hypotheses of the results. Second, we describe the used data set. Third, we present the evaluation environment.

8.2.1. Evaluation Goals

We presented approaches for all parts of a concept schema graph-based keyword search system. We focused on the efficiency. In the following, we describe the goals of the validation experiments.

Size estimators: In Section 7.1.1, we presented the query optimization and processing in the proposed system. The costs estimators are based on limited statistics. In the first set of experiments, we validate these size estimators. Since we do not use histograms or sampling techniques, we expect that the estimation is sufficient in the average case, but the cost estimation of skewed data distributions is poor. We will validate whether the average case is sufficient for the keyword search application area.

Detailed Evaluation: In a second set of experiments, we separately investigate the single steps of the keyword search processing.

Index access and query list generation: We test the time for index access and query list generation for varying keyword query sizes of plain and concept-based keyword queries. We expect this step is less expensive than to the other processing steps. Therefore, we measure the execution times as well as the query list sizes.

Materialization query generation: Query list network enumeration is the following step. We expect that the original concept schema graphs are too complex for enumeration. We propose a compact concept schema graph approach. Every concept hierarchy is represented by one complex node. Corresponding edges are compacted into one complex edge. We validate the approach by measuring the complete enumeration time of query list networks. We vary the number of plain keywords ($|Q|$) and the maximum size of query list networks $size_{max}$. Since we use the same algorithm for compact and normal concept schema graphs, the performance has to be better in the compact case. However, if we increase the complexity (more keywords, greater $size_{max}$), the compact approach shows also increasing costs, too.

Materialization query execution: The most expensive part of keyword query processing is the execution of the materialization queries. Initially, we test the all-$size_{max}$ semantics, which means the system tries to retrieve all matching object networks. We validate the empty result detection (see Section 7.2), the caching approach (see Section 7.3), and the shared plan with query list network optimization approach (see Section 7.4). We vary the number of plain keywords $|Q|$ and the maximum object network size $size_{max}$. We expect that every optimization improves the performance compared to the non-cache, non-empty result detection approach. The empty result detection will reduce the number of queries. This approach cannot reduce the number of transferred objects of successful queries. In contrast, caching will store every successful intermediate result. The problem is that the approach will re-compute empty results. Depending on the query structure, one or the other approach can be better. We expect the combination of both shows the best results. To validate these ideas, we measure the execution times, the number of executed queries, and the number of transferred local objects.

General top-k evaluation: In the general evaluation experiments, we compare the complete performance for different parameter value sets and top-k query semantics. We investigate different values for k for top-k semantics. We use the compact schema graph approach and the combination of caching and empty result detection. We argue that the top-k approach allows small query execution times. In the following, we investigate the performance of concept-based keyword queries. We measure the execution time. We expect, without query expansion,

the performance is better, if k results exist. However, we will have many empty results. Query expansion avoids this problem with a minimum overhead.

Query list optimization: For long query lists, we propose two optimizations: query splitting and query merging (see Section 7.5). Query splitting divides long condition into sub-condition and executes all combinations. This helps to detect empty results earlier. Query merging condenses queries with the same condition in one concept hierarchy into one query. It allows the efficient execution of the mediator. We expect that both optimizations improve the execution times by faster query execution and early empty result detection. In this evaluation, we implemented and tested only query splitting.

The evaluation experiments determine the execution times ($time$) as well as the main factors: number of source queries ($srcQueries$) and number of returned objects ($srcObjects$). We compare the approaches for different query parameters. The main parameters are the number of returned results k, the size of the keyword query $|Q|$, and the maximum size of results $size_{max}$ as well as the keyword parameters $KWOcc$ and DF. $KWOcc$ describes the number of occurrences of a keyword in different positions in the virtual document. DF denotes the number of objects containing a keyword in a given position. Table 8.2 summarizes the parameters.

Parameter	Description
k	the number of non-empty queries (top-k)
$\|Q\|$	the keyword query size
$size_{max}$	the maximum object network size
$KWOcc$	the number of occurrences of a keyword
DF	the document frequency of a keyword

Table 8.2.: Main evaluation parameters

8.2.2. Data Sets and Query Sets

Data sets

The Internet Movie Data Base (IMDB)[5] is a data collection about movie productions and actors. It contains information about many aspects of movies. We downloaded the data and divided the data into six databases. Figure A.1 in Appendix A.1.1 shows the concept schema of the IMDB database. Table 8.3 summarizes the statistics of the concept schema. It describes the number of concepts, data properties, and concept properties as well as the number of source databases. The number of relationships is high because we consider all combination of sub-concepts, too. These relationships are not illustrated in Figure A.1. The IMDB is a homogeneous database, and it is well usable for synthetic tests.

[5] http://www.imdb.com/interfaces, Last accessed: 2012-05-16

Data Set	Concepts	Properties	Relationships	Databases
IMDB	39	20	1024	6

Table 8.3.: Database structure

The IMDB dataset consists of six databases. Table 8.4 shows the sizes of the databases. The database `imdbperson` contains all information about persons related to movies, e.g., actors, directors, editors, and other concepts. The movie databases (`imdbmovie1,imdbmovie2,imdbmovie3`) contain the movie information and information about the positions in the movies (roles, director positions, etc.). We split the original data by using a modulo function. The database `imdbref` contains references between movies, e.g., follow ups, remakes. Finally, the database `imdbplot` contains plots of movies. Every database resides in a PostgreSQL database.

Source	Objects	Terms	Concepts
`imdbperson`	1,951,739	1,235,941	1
`imdbmovie1`	4,104,903	781,465	23
`imdbmovie2`	5,324,238	1,025,917	25
`imdbmovie3`	3,208,629	692,291	22
`imdbrefs`	431,844	26	7
`imdbplot`	110,439	204,155	1

Table 8.4.: IMDB sources

Query sets

We create different keyword query sets from the database for the different evaluation goals. As parameters of keyword selection, we use $KWOcc(kw)$ and the maximal document frequency $DF(kw)$ of a keyword. The first parameter describes in how many different positions in the virtual document a keyword occurs. It controls the number of different interpretations of a keyword. The second parameter controls the selectivity of a keyword. For example, a keyword with a high DF retrieves many objects from a source. We will describe the generated query sets in the corresponding experiments.

8.2.3. Evaluation Environment

The Yacob mediator and the keyword search component accesses the data via a Web service interface. On the server side, we use an Apache Tomcat[6] server with a Metro[7] web service stack. Client and server run on one machine that uses an AMD Phenom XII 804 with 3.2 GB RAM and Microsoft Windows XP (SP3). The Web service interface allows only conjunctive selection conditions to simulate Web data interfaces.

[6]http://tomcat.apache.org/, Last accessed 2012-03-28
[7]http://metro.java.net/, Last accessed 2012-03-28

8.3. Efficiency Evaluation

In the efficiency evaluation, we start with detailed validation of each separate step of keyword processing.

8.3.1. Size estimation evaluation

The estimation of the number of result objects and source queries is crucial to select the best plans and the best caching strategies. In the YACOB system, we use limited statistics and adapt standard cost functions. We validate the cost functions in this section with the help of the IMDB dataset and a number of exemplary queries. We expect that the estimations are reasonable and usable to the keyword search task.

For evaluation, we created nine query sets of each ten queries. Table 8.5 summarizes the statistics of the query sets. The actual query sets are presented in Appendix A.2.1. The parameters *minDF* and *maxDF* specify the document frequency interval of the selected keywords. The parameter *#bindJoins* shows the number of bind joins that every query in the set has. We want to investigate the sensitivity of the estimations with respect to the result size, the number of keywords, and the number of joins.

Query set	#Keywords	#bindJoins	minDF	maxDF
1	1	0	10	50
2	1	0	50	200
3	1	0	200	500
4	2	0	50	200
5	2	0	200	500
6	1	1	50	200
7	1	1	200	500
8	1	2	50	200
9	1	2	200	500

Table 8.5.: Query set characteristics

We executed every query set and compared the estimations of the number of queries and the number of objects with the actual values. Let W be the query set and for $q \in W$ the estimated value $est(q)$ and the actual value $act(q)$. As quality measures, we used the average absolute error

$$avgAbsErr(W) = \frac{1}{|W|} \sum_{q \in W} (est(q) - act(q)) \tag{8.1}$$

and the average relative error

$$avgRelErr(W) = \frac{1}{|W|} \sum_{q \in W} \frac{(est(q) - act(q))}{act(q)} \cdot 100\%. \tag{8.2}$$

	Queries			Objects		
Query set	*avgNr*	*avgAbsErr*	*avgRelErr*	*avgNr*	*avgAbsErr*	*avgRelErr*
4	1.0	0	0	1.5	-0.5	-21.7%
5	1.0	0	0	5.3	-4.3	-48.2%
6	364.6	0	0	605.5	26.1	19.8%
7	933.4	0	0	3101.6	-1481.8	-23.8%
8	1479.1	38.5	43.4%	1681.2	-767.2	-7.1%
9	5656.0	-232.9	51.1%	6400.2	-3132.3	1.0%

Table 8.6.: Estimation errors

Table 8.6 summarizes the results. We omitted the datasets 1 to 3 because the errors were zero, because the DF values are directly the result sizes and the number of sources was also given. For two keywords (query sets 4 and 5), the estimation underestimates the number of returned of objects because the DF values are small compared to the complete dataset size. For one or two joins, the cost functions mostly underestimate the transferred data objects. If we investigate the data in detail, in most queries the number of queries is overestimated. This can be caused by optimizations of the bind join operator that submits only distinct values. Figure 8.3 illustrates this. If the points are below the line, the values are overestimated, otherwise the numbers of objects and queries are underestimated by the system. The outliers are clearly shown. Furthermore, a detailed evaluation shows that most queries are reasonably estimated (low error values). However, some queries are outliers with high absolute errors, e.g., 30000 objects (20000 queries) (see Figure 8.2). The outliers are caused by skewed data distributions and limited statistics. For example, the result size of an actor with many roles would be underestimated. However, in average we obtain reliable estimations for keyword queries that are usable in the restricted keyword search scenario.

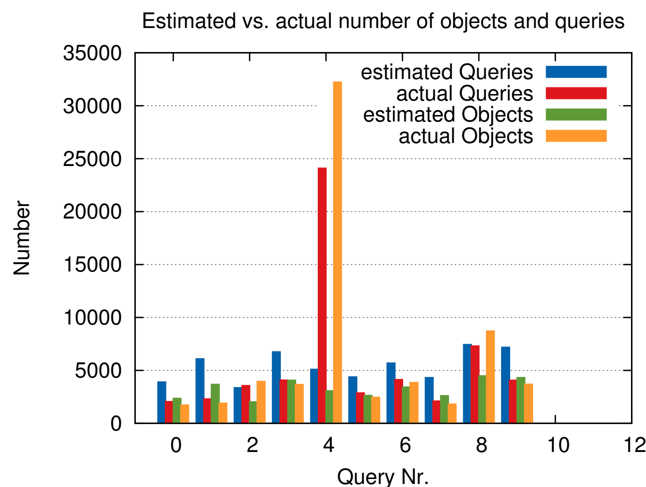


Figure 8.2.: Estimations and actual values for query set 9

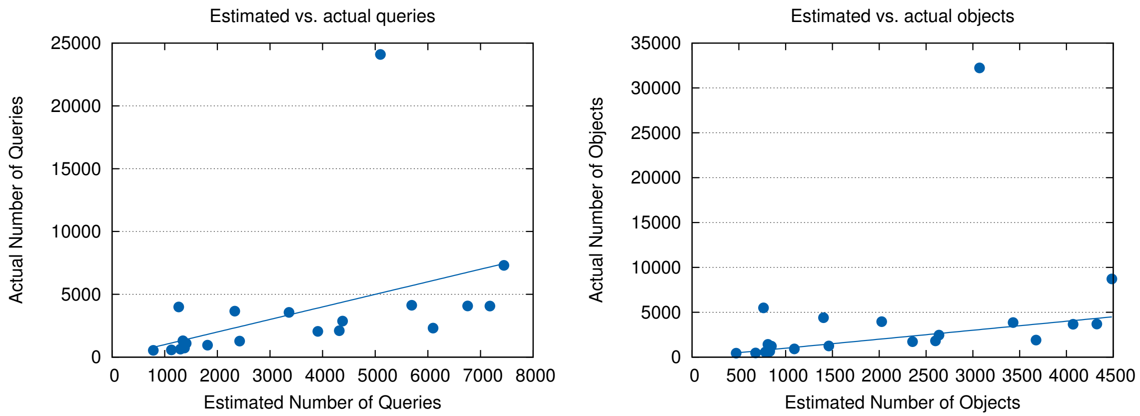


Figure 8.3.: Estimated values vs. actual values

8.3.2. Detailed Evaluation

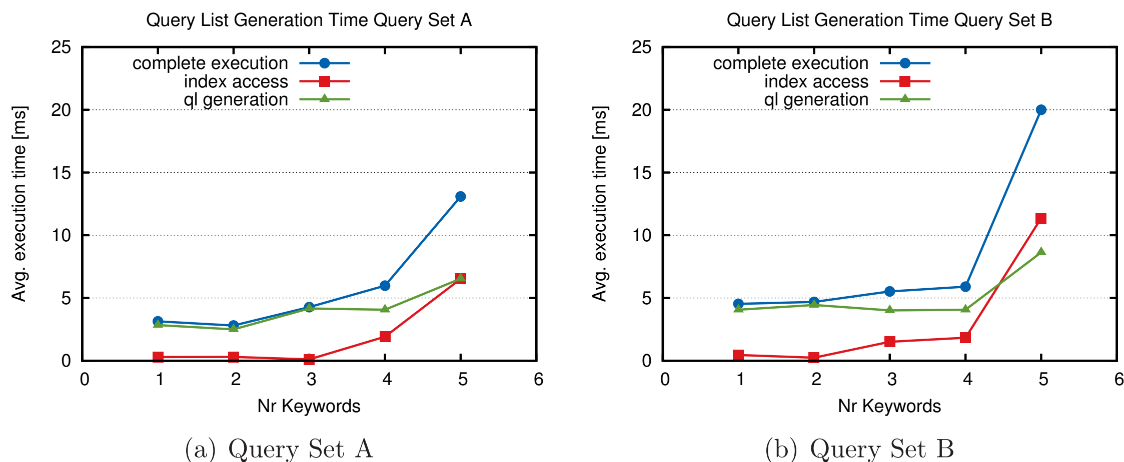
We investigate different details of the execution of keyword queries in this section. First, we provide information about index access costs and query list generation. The result of this step is an annotated concept graph. Second, we investigate the query list network enumeration. We compare the normal and compact graph enumeration. Third, we evaluate the materialization query processing algorithms.

Costs of index access and query list generation

Index access and query list generation form the first step in keyword query execution. For this, we created two plain keyword query sets A and B. Both sets consist of randomly sampled keywords that are combined to queries of different sizes. We used two different sets to determine if the sampling has an influence to the result. Both query sets vary the keyword query size $|Q|$ from 2 to 5. We outline the execution times in Figures 8.4(a) and 8.4(b). Both query sets show similar results. The index access and the query list generation times are similar and go as equal parts into the complete time. The times overall are small compared to the following steps as we will show in the remaining section.

In order to explain the execution times, we report the sizes and structures of the results. The results are single concept queries and query lists. The numbers of generated single concept queries and query lists are illustrated in Figures 8.5(a) and 8.5(b). The results show a higher number of queries and query lists for larger keyword queries because more combinations are possible. Query set B has a slightly larger result set explaining the slightly higher execution times.

The sizes of the query lists are reported in Figures 8.5(c) and 8.5(d). The average size of query lists is slowly increasing with higher keyword query sizes. However, the maximum size of a query list can be large. That happens if all keywords are supported by many different properties in one concept. In this case, the number of all combinations is high. Long query lists motivate the use of query list optimizations as described in Section 7.5.



(a) Query Set A

(b) Query Set B

Figure 8.4.: Query list generation times

Costs of query list network enumeration

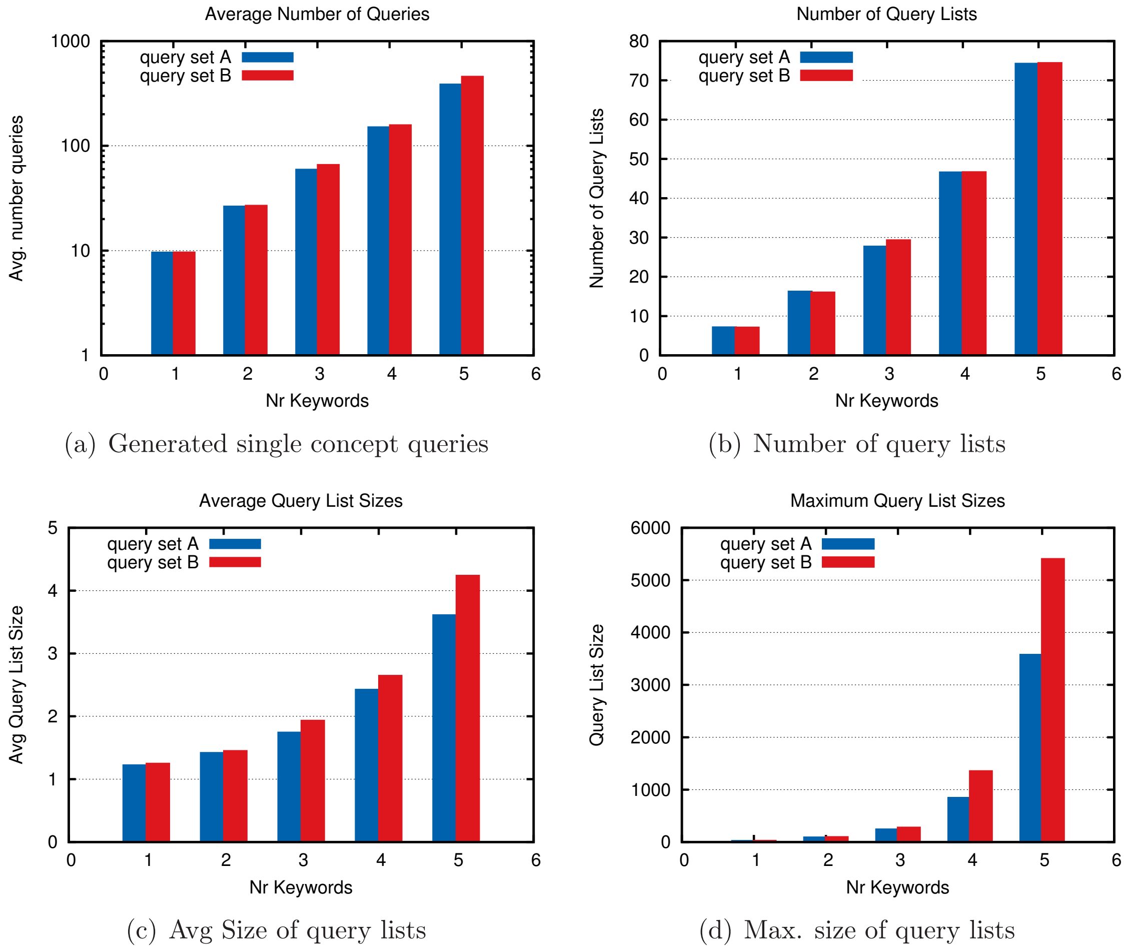
After the generation of query lists, we now compare the performances of query list network enumeration algorithms. We distinguish between the normal algorithm and the compact graph algorithm. The algorithms have the parameter maximum size of query list networks $size_{max}$. In the first experiment, we vary the parameter between 2 and 6. We compare the algorithms in Figure 8.6. The keyword query size is set to $|Q| = 2$. Because of many concepts and connections in concept graphs, the performance of the normal algorithm degrades fast in the breadth first search algorithm. The compact query graph consists of significantly fewer nodes and edges. Thus, it allows a much faster computation of query list networks. In particular, the execution times increase less with increasing $size_{max}$ in the compact case, too. This leads to much better scalability.

In the second experiment, we vary the keyword query size $|Q|$ between 2 and 5. For every keyword query size, we generated ten queries (see Appendix A.2.2). The Figures 8.7(a) and 8.7(b) show the average times for a $size_{max} = 3$ and $size_{max} = 4$. For both cases and for all query sizes, the compact graph algorithm outperforms the normal algorithm. The differences are bigger for higher values of $size_{max}$ and keyword query size $|Q|$.

In summary, the compact graph allows the efficient use of the basic breadth first search algorithm for enumerating the query list networks. Additionally to this reduction of nodes and edges, it is possible to use optimizations as described in [MYP09, Luo09, QYC11]. These algorithms avoid the checking of duplicates of query list networks. The combination with the compact graph approach can improve the performance further.

Costs of query execution

In the following experiments, we investigate the query execution performance for all-$size_{max}$ semantics. That means that we require all results for a plain keyword query. The input is a list of query list networks. For generation of materialization queries from



(a) Generated single concept queries (b) Number of query lists

(c) Avg Size of query lists (d) Max. size of query lists

Figure 8.5.: Number of generated single concept queries and query lists

the query list networks, we use the algorithms presented in Section 7.1.2: complete query generation and the step-by-step algorithm by Hristidis et al. [HGP03]. Both are similar in performance, but the second is more memory efficient as it creates materialization queries on the fly. As compact concept graphs are shown to be more efficient, we assume the query list networks are generated using the compact graph algorithm.

We investigate different plain keyword query sets in the experiments. We distinguish the keywords by the number of occurrences in the global virtual document, i.e., in how many concepts and properties a keyword is found. We denote this parameter as $KWOcc$. We created three sets of keywords with 1 to 4, 5 to 10, and 10 to 15 occurrences. We created from each of these sets plain keyword queries that are not empty. Furthermore, we varied the number of keywords $|Q|$ from 2 to 4. The query set is analyzed in Appendix A.2.3. The number of maximum object network sizes $size_{max}$ ranges between 3 and 5. The goal of the experiment is to validate every optimization method proposed in Chapter 7.

We report the execution times for every optimization method. Table 8.7 summarizes the studied algorithms.

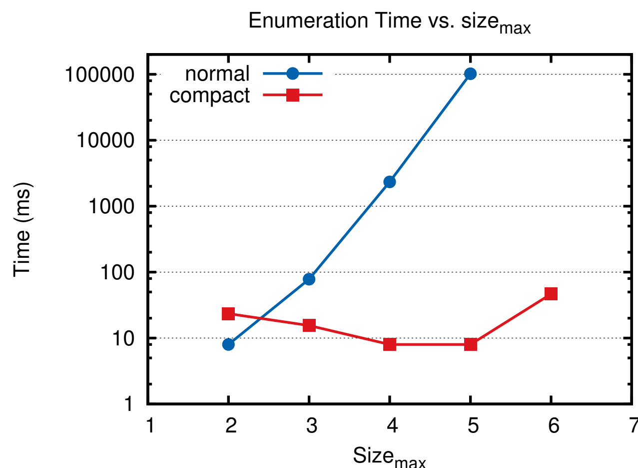
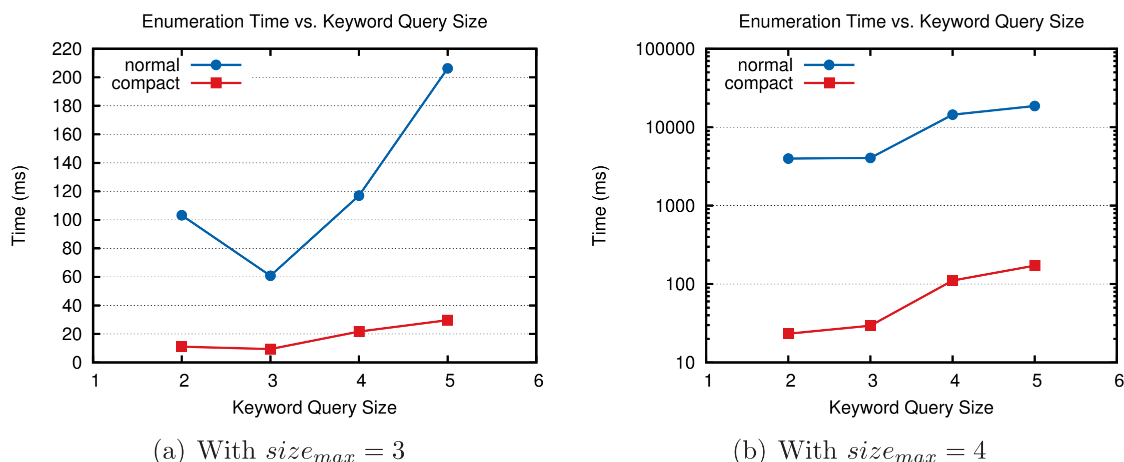


Figure 8.6.: Query list enumeration: $size_{max}$ vs. time



(a) With $size_{max} = 3$

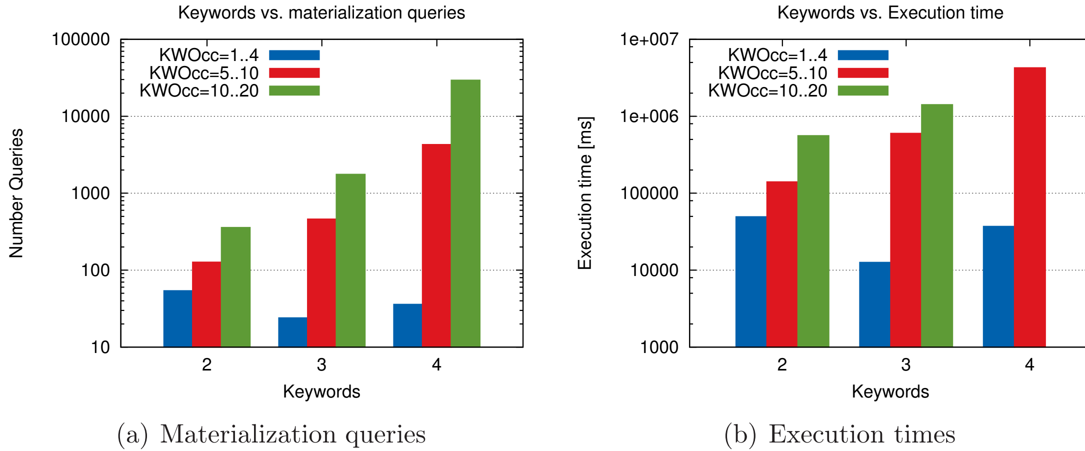
(b) With $size_{max} = 4$

Figure 8.7.: Query list enumeration: keyword query size $|Q|$ vs. time

Unoptimized results. Initially, we show the influence of the parameters $KWOcc$, the number of keywords $|Q|$, and the maximum network size $size_{max}$. We report the results for $2 \leq |Q| \leq 4$ and $size_{max} = 3$ in Figure 8.8. Figure 8.8(a) shows the number of materialization queries for every keyword query size. Generally, the number increases with keyword query size and $KWOcc$. However, the keyword query with two keywords with $KWOcc = 1..4$ has a higher number of occurrence than the queries with more keywords (see Appendix A.2.3). Therefore, the query has higher costs. Figure 8.8(b) shows the resulting execution times. The time values increase with the number of materialization queries, because the document frequencies of the keywords are similar. We omitted the execution time of the last query ($m = 4, KWOcc = 10 \dots 20$) because of excessive runtime. The data shows that the query execution time increases exponentially. In Figure 8.9, we show the influence of the $size_{max}$ parameter. With increasing $size_{max}$, the number of generated materialization queries and the execution time are increasing exponentially. After showing the influence of

Optimization	Description
er	empty result detection
cache	semantic cache approach
er + cache	empty result detection + semantic cache
cache_adv	shared plan
er + cache_adv	empty result detection + shared plan

Table 8.7.: Compared algorithms



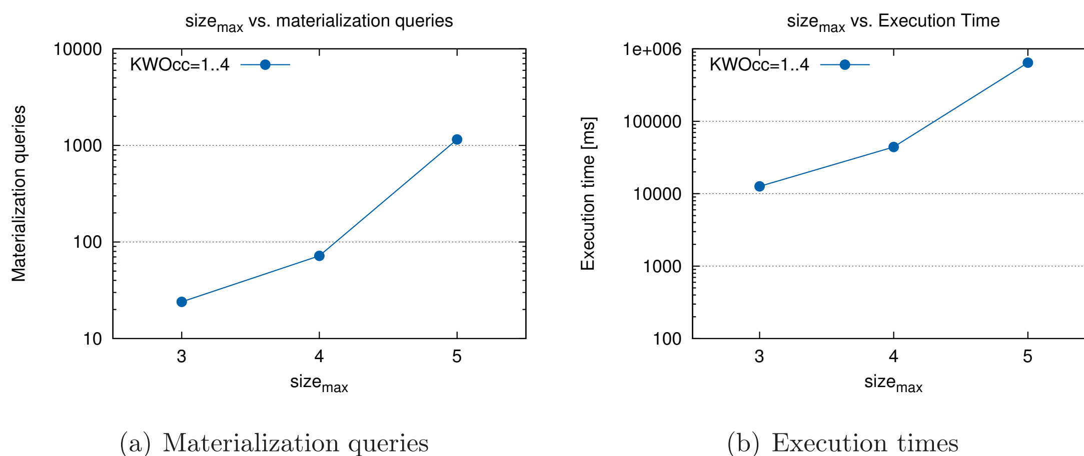
(a) Materialization queries (b) Execution times

Figure 8.8.: Influence of the keyword occurrence $KWOcc$ on the execution time for $size_{max} = 3$

the parameters, we now test the optimization approaches. Figure 8.10 compares the execution times of the different approaches.

Figure 8.10(a) illustrates the influence of the keyword query size. The results show that optimizations reduce the execution times. It also shows that empty result detection and caching are necessary. Shared plans improve the performance but not in every case. The results indicate the combination of empty result detection and caching leads to the best results. Figure 8.10(b) shows the results for varying $size_{max}$ and $|Q| = 3$. It shows that the fully optimized execution times increase slower than the unoptimized. Finally, the last test shows that the costs are increasing with increasing $KWOcc$ because more materialization queries are constructed and executed. The optimizations showed a higher benefit for expensive queries.

To explain the result, we present the number of executed source queries and of transferred objects in Figure 8.11. Figures 8.11(a), 8.11(c), and 8.11(e) show the number of source queries. The numbers are reduced by empty result detection and caching approaches. On the one hand, the caching approaches cannot avoid empty queries. Thus, they execute a large number of not necessary source queries. On the other hand, the empty result detection only algorithm has to execute repeatedly successful queries. This leads to a higher number of transferred objects as illustrated in the Figures 8.11(b), 8.11(d), and 8.11(f). The caching approaches cause the re-computation of empty results. This leads to many queries without results. In consequence, we need



(a) Materialization queries (b) Execution times

Figure 8.9.: Execution times with respect to the parameter $size_{max}$ for $|Q| = 3$

both cache and empty result detection to reduce the number of source queries and of source objects, but also to reduce the load and space used on the global level.

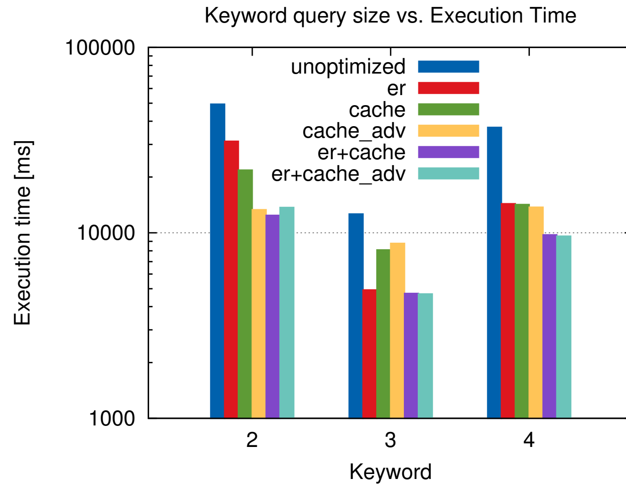
The results also show that the number of source queries and source objects can be reduced to reasonable numbers. However, the execution times are still high which makes the all-$size_{max}$ semantics to an exploratory approach of possible results. The semantics is not well suited for ad-hoc queries. However, with application of further optimizations on the global level as proposed in the related systems [QYC09, BRL$^+$10], the execution times are further reduced.

8.3.3. Top-k evaluation

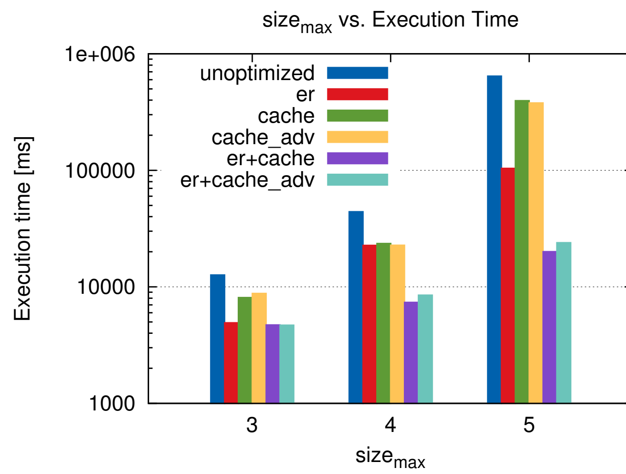
We now evaluate top-k query semantics for plain and concept-based keyword queries. We investigate the influences of the parameters k, $|Q|$, and $size_{max}$. We use the IMDB dataset and generate various query sets. During the evaluation, we use the empty result detection and shared plan algorithm, because it showed the best performance in the previous experiments for all-$size_{max}$ semantics. We start with plain keyword queries.

Plain keyword queries

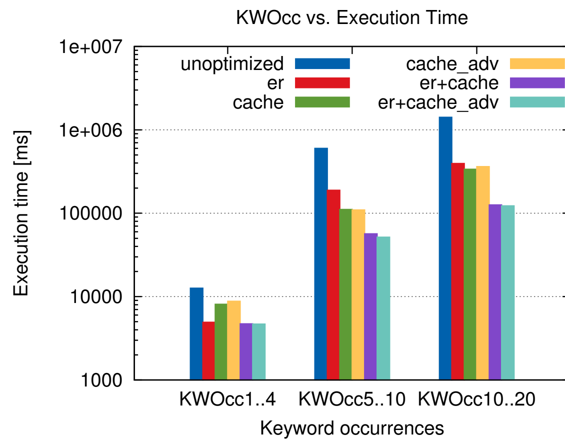
We create three sets of plain keyword queries with 2, 3, and 4 keywords. The query sets are described in Appendix A.2.4. We selected the queries, such that, they have at least ten results. We report the results for varying k in the Figures 8.12 and 8.13. First, we use queries with two keywords and vary $size_{max}$ between 2 and 6. The average execution time increases with $size_{max}$ and k. However, for top-1 queries the execution time is constant, because similar top-1 results are found for each $size_{max}$. For $size_{max}$ values of 5 and 6, the same top 5 and 10 values have been found. Therefore, we report similar execution times. As we used queries, which have at least ten results, the result has a bias. In particular, if a query has less than k non-empty queries, the system will execute all materialization queries. In this case, the performance significantly degrades.



(a) Optimization vs. keyword query size, $size_{max} = 3$ and $KWOcc = 1..4$

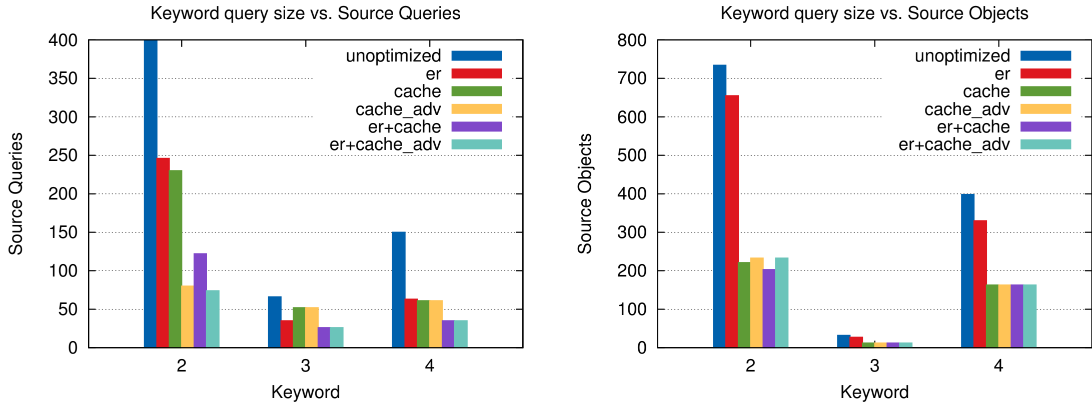


(b) Optimization vs. $size_{max}$, $|Q| = 3$ and $KWOcc = 1..4$



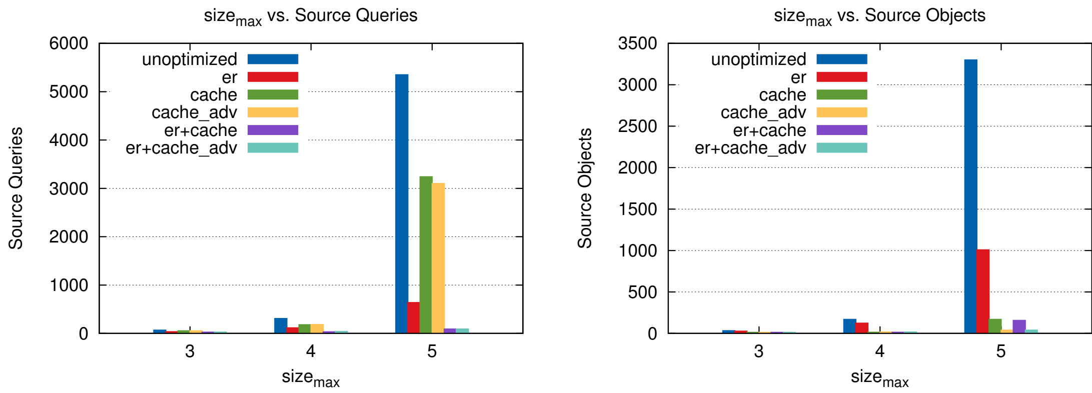
(c) Optimization vs. $KWOcc$, $|Q| = 3$ and $size_{max} = 3$

Figure 8.10.: Optimization vs. execution time



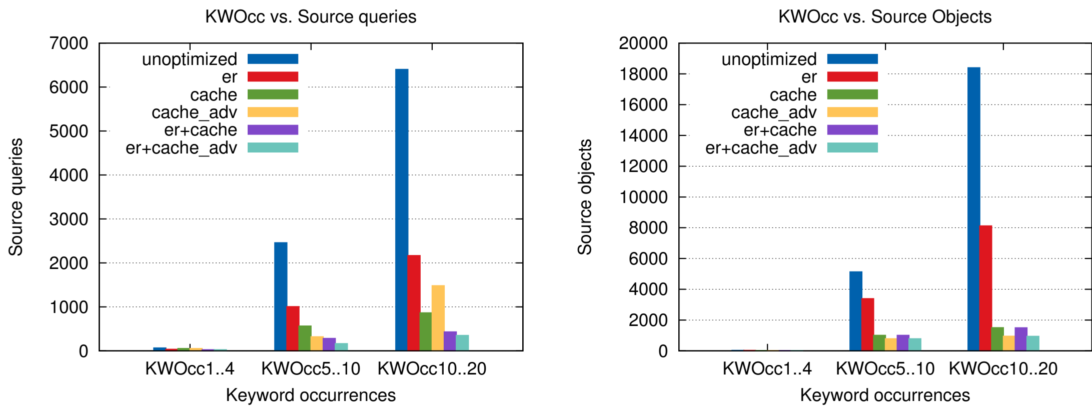
(a) Optimization vs. Keyword query size, $size_{max} = 3$ and $KWOcc = 1..4$

(b) Optimization vs. Keyword query size, $size_{max} = 3$ and $KWOcc = 1..4$



(c) Optimization vs. $size_{max}$, $|Q| = 3$ and $KWOcc = 1..4$

(d) Optimization vs. $size_{max}$, $|Q| = 3$ and $KWOcc = 1..4$



(e) Optimization vs. $KWOcc$, $|Q| = 3$ and $size_{max} = 3$

(f) Optimization vs. $KWOcc$, $|Q| = 3$ and $size_{max} = 3$

Figure 8.11.: Optimization strategies vs. execution time

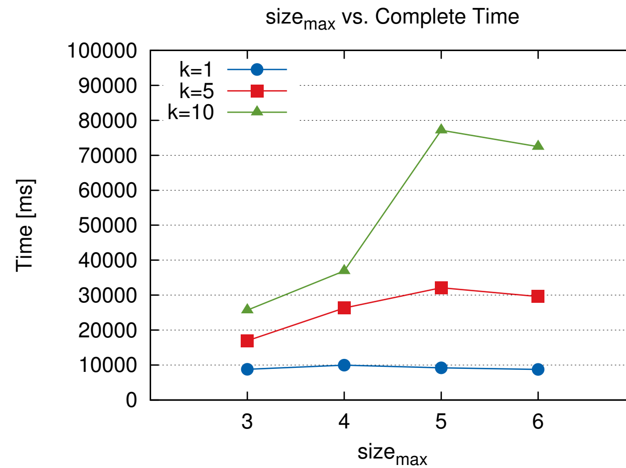
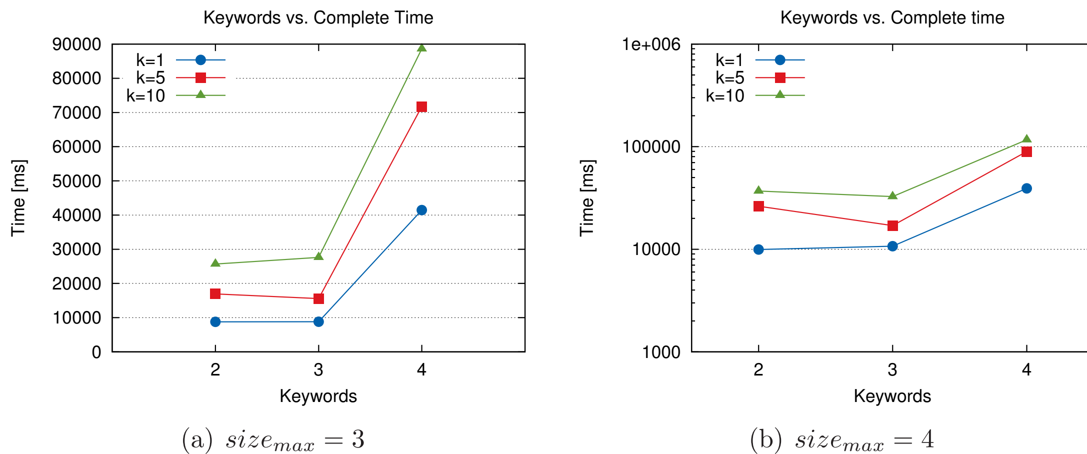


Figure 8.12.: Top-k overall execution times for different $size_{max}$ and $|Q| = 2$

Figure 8.13 shows the results for varying keyword query sizes. It shows that the performance is quite similar for $|Q| = 2$ and $|Q| = 3$. The performance starts to worsen for more keywords and higher k values.



(a) $size_{max} = 3$ (b) $size_{max} = 4$

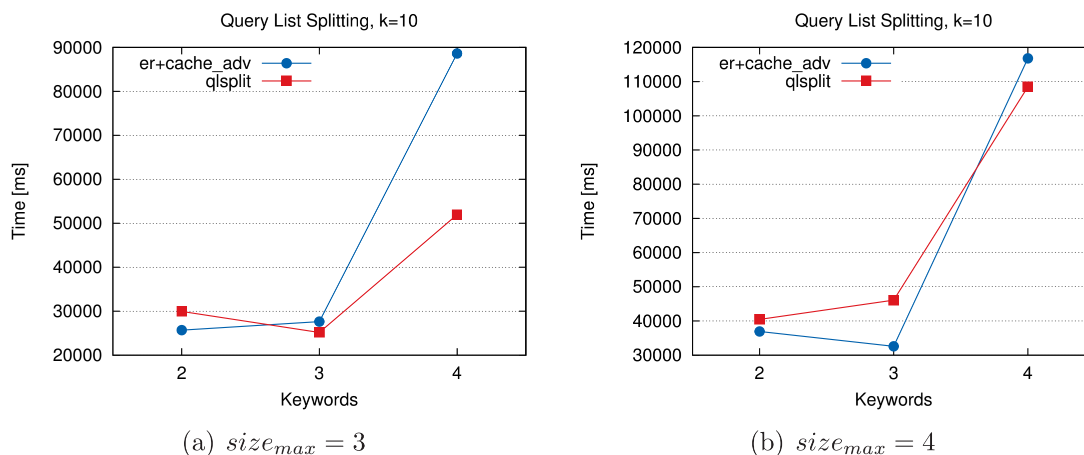
Figure 8.13.: Top-k overall execution times for different keyword sizes

In summary, the queries are executed in 10 to 100 seconds. For top-1 queries, we have reasonable times. Higher keyword counts and $size_{max}$ values increase the execution times, which are significantly lower than queries in the all-$size_{max}$ semantics.

Query splitting optimization

If keyword query contains many terms, the system can create long query lists for single concepts. In particular, many combinations of keywords will not create results. Thus, the query list optimizations can reduce the number of queries sent to the sources. Therefore, we developed the query list splitting optimization. That means that a conjunctive condition is split into sub-conditions. Queries with the sub-conditions are

executed. The results are combined in further steps. In this way, intermediate empty results are easily discovered. We assume keyword queries with more keywords and smaller $size_{max}$ benefit more from query list splitting. Figures 8.14(a) and 8.14(b) illustrate results. The results validate the assumptions. In detail inspection of the results, one can see that query splitting allows the detection of more empty results. However, the approach is better with lower k values. During the evaluation, the proposed scoring function favors small results with many keywords in few concepts. That situation creates long query lists.



(a) $size_{max} = 3$

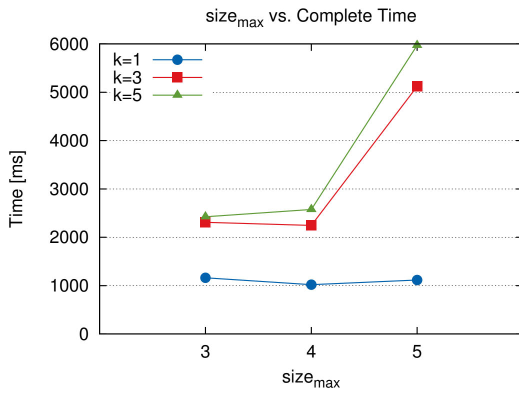
(b) $size_{max} = 4$

Figure 8.14.: Top-k query list splitting

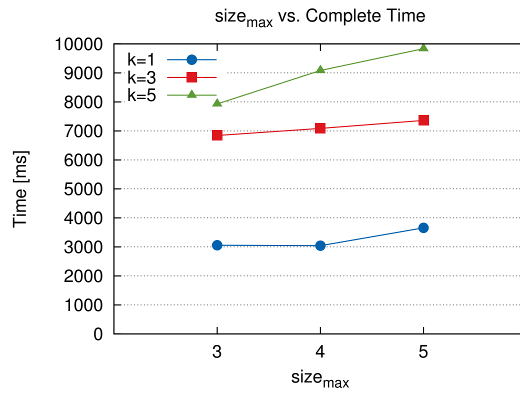
Concept-based keyword queries

Plain keyword queries do not give any constraints for the positions of the keywords. However, if the user has limited knowledge about the position of a keyword, concept-based queries are advantageous. In the case of wrong concept-labels, query expansion helps to mitigate the problem. We sample three query sets with two, three, and four concept-based query terms (see Appendix A.2.5). In this section, we report the execution times for these queries. Figure 8.15 shows the execution times for concept-based keyword queries and expanded concept-based keyword queries of the size $|Q| = 2$ and varying $size_{max}$ and k. The execution times are lower than for plain keyword queries because of a lower number of generated queries. However, a number of non-expanded keyword queries could not generate results. They could not generate valid query list networks. With the help of query expansion, we could create results. The performance of query times of 1 second to 9 seconds is low compared to plain keyword queries.

In a second experiment, we try to evaluate the influence of the size of keyword queries. The results are reported in Figure 8.16(a) for concept-based keyword queries and in Figure 8.16(b) for expanded concept-based keyword queries. Bigger queries seem to have a better performance. However, they also have more often empty results, because we cannot build many combinations using concept-based keyword queries.



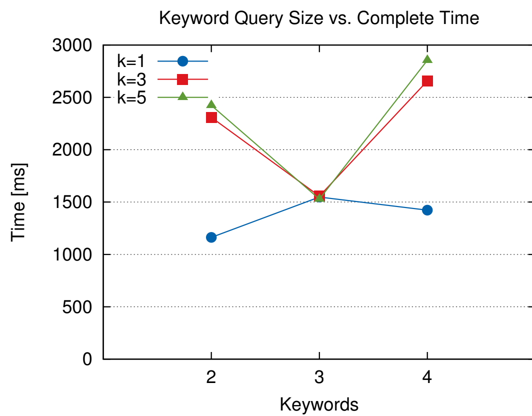
(a) Normal



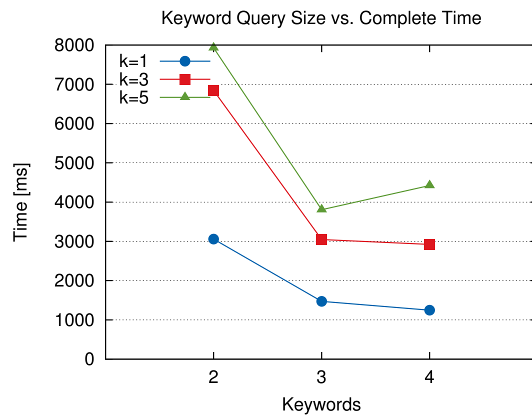
(b) Concept expanded

Figure 8.15.: Top-k complete execution times for different $size_{max}$ and $|Q| = 2$ of concept-based keyword queries

The positions of keywords are specified. This leads to a better performance. However, possible unexpected and relevant results might be missed.



(a) Normal



(b) Concept expanded

Figure 8.16.: Top-k complete execution time vs. concept-based keyword size, $size_{max} = 3$

8.4. Effectiveness Evaluation

Effectiveness of a search system describes how good the results are in relevance to the query. Here, we want to compare plain and concept-based keyword queries. For this, we created sets of keyword queries and added concept and property labels. After that, we compare the best 15 results. Table 8.8 outlines the queries. The first set searches for connections between persons of the name “ledger” and “gyllenhaal”. Expected are

Set	No.	Query
1	1	ledger, gyllenhaal
	2	person::ledger, person::gyllenhaal
	3	:name:ledger, :name:gyllenhaal
	4	person:name:ledger, person:name:gyllenhaal
2	5	sci-fi:title:trek,person:name:stewart
	6	drama:title:trek,person:name:stewart
	7	trek,stewart
	8	sci-fi:title:star,sci-fi:title:trek,person:name:stewart
3	9	person:name:kubrick,movie:title:strangelove,person:name:sellers
	10	:name:kubrick,:title:strangelove,:name:sellers
	11	kubrick,strangelove,sellers

Table 8.8.: Query set for the effectiveness experiment

the connections between Heath Ledger and Jake[8] or Maggie Gyllenhaal[9], respectively. The maximum result size was set to $size_{max} = 5$. The inspection of the results (see Appendix A.2.6) reveals, the best results are obtained with query 4 that exactly specifies the position of the keywords. The results of the queries show exactly the expected connections. Queries 1 and 2 show mostly queries to the concept person because both keywords "ledger" and "gyllenhall" occur in different properties of concept "person". In this case, the property label is more powerful than the concept label. This shows that both labels are justified.

The second set of queries tests the query expansion of labels. We search for a connection between the keywords "trek" and "stewart". The results show that concept label expansion can effectively improve the results because the wrong label "drama" did not influence the results. Furthermore, the query set 2 confirms that the proposed ranking function favors small results. One solution is the concept network query semantics. Here, we would get all possible connections mitigating this problem. A second conclusion of query set 2 is that the developed solution must be extended to phrase search. It is hard to express the expression "Star Trek" as query 8 illustrates. Even if the expected results are returned, phrase search will improve the performance, too.

Query set 3 is used to confirm the previous results. Query 9 showed the best effectiveness. The result set of Query 10 reveals that the property label alone is not sufficient, if the properties occur in different concepts. For example, "name" applies to the concepts "Person" and "CharacterInMovie".

In this study, we focused mainly on efficiency. This effectiveness study uses only exemplary queries and analyzes the result sets. In order to assess the quality of effectiveness, we have to provide a user study and measurements like precision and recall. We will do this in future work. However, the small study confirmed that

[8]Played together in the movie "Brokeback Mountain"
[9]Played together in the movie "Dark Knight"

- concept-based keyword queries help to improve effectiveness and efficiency,
- the proposed ranking function shows promising results but must be optimized to avoid large numbers of single concept results, and
- concept-network query semantics helps to improve effectiveness by avoiding many single concept results.

8.5. Summary

In this chapter, we presented the prototypical implementation and a set of experiments based on the IMDB dataset. The implementation uses the YACOB mediator prototype and extends it by keyword search components. We focus in this chapter on the online phase of the keyword search system. The main focus lied on efficiency studies. Initially, we validated the proposed basic cost estimation function. As the functions and statistics are limited, the experiments showed that extreme cases and skewed data cannot be estimated, but, in average, the estimation is appropriate. This is in line with related studies.

In general, the evaluation confirmed that the execution phase dominates the overall costs. Experiments of the keyword index showed query times of 10 milliseconds. Optimization can improve the performance. The materialization query enumeration experiments confirmed that compact concept graphs significantly improve the performance.

The execution experiments provided further insights in the most expensive part of keyword query processing. They confirmed that both, empty result detection and caching, are necessary to improve the performance. Furthermore, they showed that query expansion of concept-based keyword queries does not impose higher costs, but improves the results.

The effectiveness study was limited, but confirmed the assumption that concept-based keyword queries improve the query result quality. Furthermore, it revealed that phrase search should be supported.

In summary, the studies showed that keyword search across heterogeneous structured databases using a concept-based mediator are possible but expensive. Further optimizations are:

- a phrase search that avoids many keywords,
- application of optimizations on global level after most of the data is materialized,
- better cost functions and statistics to improve the estimations, and
- use of the concept-network query semantics.

9. Summary and Future Work

Nowadays, large numbers of semi-structured and structured data sources are available in the Internet and Intranets. Many information needs of users and applications can only be satisfied by integrating several, heterogeneous sources. As structural heterogeneity is high, concept-based or semantic integration systems have emerged in the last decades. They annotate local data with concepts of a concept schema to enable the connection of information from different sources. Concept-based integration systems offer a uniform, but complex access to the data, like concept-based query languages or browsing.

Centralized (semi-)structured database systems also have complex schemata and query languages. Thus, in the last decade keyword search systems over XML, relational, and graph databases have been developed [YQC10]. In contrast to document collections, keyword search over structured sources has additional challenges because keywords and information are distributed over different data elements like XML data items, tuples, relations, or objects. Therefore, keyword search systems have to find connections between these elements. While many approaches exist that support centralized approaches, virtually integrated, structured data sources are less supported. Notable exceptions are the selection of relational databases [YLST07, VOPT08] and keyword queries across heterogeneous relational databases [SLDG07].

This thesis develops and evaluates concepts for the combination of concept-based mediator systems and keyword search over the concept-based data model. The result is a keyword search system that supports the search across heterogeneous, semi-structured data sources. The underlying mediator system Yacob provides a uniform access to sources and ensures the integration of local objects to global objects. Furthermore, it allows the efficient execution of single concept queries. The developed keyword search system uses the mediator system and a global keyword index to provide the keyword search and join functionality. The system follows the schema-graph approach and generates concept-based query statements, denoted as materialization queries. The join processor optimizes and executes materialization queries. We use query coverage and query containment to avoid re-computation of empty results and to allow re-use of intermediate results, respectively. Both optimizations provide a performance gain as the evaluation validates.

The thesis shows that it is possible to create the combination of mediator and keyword search to simplify the search across heterogeneous structured data sources. The results also show that keyword queries are expensive to process, making the search mostly an explorative task. However, optimizations mitigated the high costs. The use of labeled keyword queries, denoted as concept-based keyword queries, combined with query expansion, improves performance in efficiency and effectiveness because the system can rely on users query hints.

9.1. Summary

Chapter 2 and Chapter 4 provide the background of concept-based mediator systems and keyword search over structured data sources, respectively. Furthermore, both chapters classify and compare the respective related work. Both chapters complement existing surveys on the respective systems [HRO06, WVV+01, YQC10, PgL11] by focusing on the topics of this thesis: concept-based integration and keyword search across of distributed sources. In particular, Chapter 4 discussed keyword search over heterogeneous, virtually integrated sources.

Chapter 3 describes the concept-based mediator system YACOB. The YACOB system was introduced by Sattler et al. [SGHS03]. Subsequent works extend the system by keyword search in single concepts [Dec04, Gei04] and a semantic cache [Kar03, KSGH03]. Sattler et al. provide the complete system overview [SGS05]. Chapter 3 improves and extends the definitions and specifications of YACOB. The YACOB system provides the integration service, the domain modeling, and the query planning for single concept queries. It is the basis of keyword search described in the remainder of the thesis.

Chapter 5 defines concept-based keyword queries. We assume a concept-based schema graph and a description of the actual data in the form of a keyword index. The keyword search definitions abstract from the YACOB model but use its main features: concepts and properties, concept hierarchies, category hierarchies for conceptual data values, and mapping information. This information is also featured by many other integration systems. Based on this information, we define the Virtual Document and concept-based keyword queries. As results of keyword queries, we consider two steps. First, we see materialization queries as interpretations of keyword queries. Second, object networks are the results of materialization queries and the final results of the keyword queries. Results are ranked by the score of materialization queries. Concept-based keyword queries are labeled keywords. In order to mitigate wrong concept labels, we introduce query expansion for keyword queries. Concept label expansion exploits the concept hierarchies and category keyword expansion exploits category hierarchies. Finally, we propose three query semantics: all results, top-k non-empty queries, and top-k concept networks. For top-k semantics, a ranking function is defined. The function combines schema level and data level scores as well as the compactness of results and the size of the object networks. The ranking function also includes concept label expansion. However, the function focuses on showing the applicability and efficiency. The function is not optimized and validated for effectiveness. As the ranking function is monotonic, it allows straightforward but efficient top-k algorithms.

Chapter 6 outlines our keyword search solution. We propose a schema graph-based evaluation approach [HP02]. We use the concept schema as schema graph. Because of concept hierarchies and many concept-properties, the schema graph is highly complex. It is necessary to compact the graph to reduce its complexity. The chapter includes all steps to build materialization queries: the keyword index structure, index

lookup methods, the single concept query generation, and the generation of query list networks. The keyword index and single concept generation algorithms are adapted from [Dec04, GDSS03] and improved. Query list networks are compact representations of a set of materialization queries. They are the input of the next step: result creation.

Chapter 7 is concerned with the efficient processing of materialization query sets. It adapts existing approaches of tuple and query sets processing [HGP03, ZZDN08] to query processing. It emphasizes the importance of avoiding unnecessary source queries. In particular, bind join operations are expensive operations. Thus, we discuss three approaches to reduce query costs: empty result detection, a semantic cache, and materialization of intermediate results. One result is the definition of query coverage and query containment for concept-based queries. Furthermore, we show how to optimize a query list network at whole. At last, we provide ideas for query merging and query list splitting.

Chapter 8 describes the system architecture and implementation. Different experiments validate the algorithms defined in the previous chapters. The experiments comprise the general execution of keyword queries, but also detailed investigations of the different parts of the keyword query process. We use different query sets and vary fundamental parameters for the evaluation tests.

9.2. Contributions

The contributions of the thesis are grouped into three areas of topic: concept-based integration, concept-based keyword query definitions, and execution of keyword queries over concept-based models.

Concept-based integration and query processing. The Yacob system is a member of the group of concept-based mediator systems. It allows the integration of Web sources and other kinds of semi-structured sources. The Yacob system allows the Local-as-View source mapping of XML sources to a concept-based model. Additionally to the published work of Sattler et al. [SGHS03, KSGH03, GDSS03, Kar03, Dec04, Gei04, SGS05], this thesis improved the description of the Yacob system. In order to support efficient keyword queries, we provided join processing based on semi-joins and bind-joins. The use of semi-joins and subsequent global joins reduces the number of materialized tuples and allows a seamless integration of a concept-based semantic cache. The materialized semi-join results is better reused by similar queries.

Concept-based keyword query definitions. We defined concept-based keywords as an instance of labeled keywords. We provide a possibility to search heterogeneous sources using all information: schema terms, information from source descriptions, and elementary content descriptions. We support thereby also concept-based query expansion. We adapt the semantic relationship in that way that the expanded terms

are intuitive according to concept-schema, i.e., concept subClassOf relationship. This allows effortless query formulation. The inclusion of mapping information allows users to utilize their knowledge about local sources. Providing concept labels and query expansion allow the efficient and effective usage of partial knowledge of the concept schema.

Schema graph-based evaluation of keyword queries Schema graph-based evaluation of keyword queries creates many overlapping materialization queries. As we can control the join processing in our system, we contribute the following points to exploit the overlapping:

- detection of empty results for concept-based queries with keyword containment predicates to avoid unnecessary re-computations,
- semi-join results as cache and intermediate results to avoid re-computations of expensive bind-join queries and to reduce the overhead of stored tuples, semi-joins are also exploited in relational databases [QYC09] and relational data streams [QYC11],
- query list optimization to synchronize queries in order to maximize the re-use of the cache, and
- a shared plan of materialization queries by step-wise addition of query plans.

Another point is the definition of concept-based keyword queries with query expansion. User hints allow the reduction of query costs. The proposed optimization approaches can be translated to every keyword search system over structured data.

9.3. Future work

Based on the results of this work, we propose several extensions and possible research directions.

Query language. The evaluation of the system showed that phrase search has to be supported. For example, the query `painter:name:holbein,painter:name:younger` causes an execution overhead and even returns unnecessary results. A solution is phrase search. The corresponding query is `painter:name:"holbein the younger"`. This query indicates the keywords have to occur always in a common property value. Phrase search can be supported by modifying the index lookup. Index entries for the value keywords have to have common concept and property values and all keywords of the phrase have to be in exactly one value.

Ranking function effectiveness. This work focuses on the efficiency of keyword queries. It uses the proposed ranking function to illustrate the top-k functionality and to include query expansion distances. The ranking function is not optimized for effectiveness. Thus, one has to optimize the ranking function by considering different

approaches. First, it is possible to improve the term weight definition following [Sin01]. Second, different ranking functions, also non-monotonic, have to be tested, for example, page-rank inspired functions. Third, one can include more information like average object value lengths. Fourth, the ranking model should be taken to a consistent model based on the vector space or probabilistic model.

Ranking of objects and object networks. This work does not compute the rank of objects and object networks. Instead, we rank results only by query scores. However, queries might have more than one result and instance scores can be different to the query scores. This is equivalent to the ranking of documents in distributed information retrieval [Cal00]. The problems are manifold. One has to merge the local scores to a global score. It is necessary to combine materialization query score and object network score. For example, Xu et al. ranked candidate networks, first, and then tuple trees for improving the effectiveness of keyword search in relational databases [XIG09]. From this follows further research directions, like ranked cache results and ranked joins. One can adapt ideas like [MBG04] as a starting point for object networks to optimize the results and reduce the number of retrieved objects. Ranking object networks has a significant potential to improve top-k queries, because we do not have to retrieve all objects but only the necessary objects for a top-k result.

Query optimization. We proposed in this work a basic cost model. It has the advantage of using only a limited set of statistics. The statistics are generated directly from the keyword indexes. The disadvantage is the problem of skewed data distributions in the join computation (see Section 8.3.1). Furthermore, the number of queries and objects are only two parameters. Thus, it is advantageous to consider more sophisticated statistics like histograms, on the one hand. On the other hand, one has to include further optimization parameters like response time of sources for a query and the global load. Ideally, the necessary statistics structures should be self-learning and self-maintaining.

Keyword search support. The keyword indexes in this work are single term indexes. That means that we do not store information whether keywords are connected assuming a certain distance. If we would have these connection information between keywords, we could reject keyword queries directly. A possible solution is based on keyword relationship matrices [YLST07] or keyword relationship graphs [VOPT08]. Both solutions are used to select the best relational databases for multi-database keyword search, i.e., results across different sources are not supported. Thus, the problem arises that connections across different sources have to be created without materialization of the complete integrated dataset. A solution could be a self-learning method. We can also add non-connected keywords for a certain distance. In the first step, the empty result statistics is a source. In a second, we compress the statistics to keyword non-connections. Equivalently, we can add results of keyword queries to a connection index. These self-tuning indexes can complement source descriptions with keyword connections.

Interactive keyword queries. Chu et al. [CBC+09] and Demidova et al. [DZN10] proposed the combination of form based search and keyword search as well as the interactive generation of queries, respectively. Particularly, it is attractive to adopt these approaches to our keyword search system because expensive source queries cause high running times. We already create materialization queries as an intermediate step. As many similar materialization queries are generated, one has to find grouping approaches, for example, based on common concept networks. From that starting point, the user can select a group of materialization queries, create labeled keyword queries, or combine both methods.

Similarity joins and learned mappings. Another future extension is the inclusion of inexact mappings and join conditions. On the one hand, it is possible to add similarity joins as concept property mappings [Sch04, SGS04]. In the KITE system [SLDG07], similarity joins are used. However, the system supports only relational database systems and cooperative systems. On the other hand, we could use learned mappings from sources to the global concept schema. The mappings are not exact but have a given quality. The mapping quality must be included in the ranking function of the keyword queries.

Virtually integrated graph databases. In this work, we focus on the schema graph-based keyword search approach. However, instances of a concept schema form a data graph. Thus, it is worthwhile to investigate how to deal with a distributed, heterogeneous data graph and keyword search. Existing data graph-based approaches [HN02, KPC+05, LFO+11] have to be adapted to cope with overlapping graphs that are connected across different, heterogeneous sources. One possible application of this extension is distributed open-link data.

Bibliography

[AAB+98] José Luis Ambite, Naveen Ashish, Greg Barish, Craig A. Knoblock, Steven Minton, Pragnesh Jay Modi, Ion Muslea, Andrew Philpot, and Sheila Tejada. ARIADNE: A System for Constructing Mediators for Internet Sources. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 561–563. ACM Press, 1998.

[ABFS02a] Bernd Amann, Catriel Beeri, Irini Fundulaki, and Michel Scholl. Ontology-Based Integration of XML Web Resources. In Ian Horrocks and James A. Hendler, editors, *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, volume 2342 of *Lecture Notes in Computer Science*, pages 117–131. Springer, 2002.

[ABFS02b] Bernd Amann, Catriel Beeri, Irini Fundulaki, and Michel Scholl. Querying XML Sources Using an Ontology-Based Mediator. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 429–448. Springer Berlin / Heidelberg, 2002.

[Abi97] Serge Abiteboul. Querying Semi-Structured Data. In Foto N. Afrati and Phokion G. Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1997.

[ACD02] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*, pages 5–16. IEEE Computer Society, 2002.

[ACD+03] B. Aditya, Soumen Chakrabarti, Rushi Desai, Arvind Hulgeri, Hrishikesh Karambelkar, Rupesh Nasre, Parag, and S. Sudarshan. User Interaction in the BANKS System. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 786–788. IEEE Computer Society, 2003.

[ACPS96] Sibel Adali, K. Selçuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 137–148. ACM Press, 1996.

[AHK97] Yigal Arens, Chun-Nan Hsu, and Craig A. Knoblock. Query Processing in the SIMS Information Mediator. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 82–90. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[AK93] Yigal Arens and Craig A. Knoblock. SIMS: Retrieving and Integrating Information From Multiple Sources. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.*, pages 562–563. ACM Press, 1993.

[AKS96] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query Reformulation for Dynamic Information Integration. *Journal of Intelligent Information Systems (JIIS)*, 6(2/3):99–130, 1996.

[AKS99] Naveen Ashish, Craig A. Knoblock, and Cyrus Shahabi. Selectively Materializing Data in Mediators by Analyzing User Queries. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems, Edinburgh, Scotland, September 2-4, 1999*, pages 256–266. IEEE Computer Society, 1999.

[AKYJ03] Shurug Al-Khalifa, Cong Yu, and H. V. Jagadish. Querying Structured Text in an XML Database. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 4–15. ACM, 2003.

[APTP03] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Scalable template-based query containment checking for web semantic caches. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 493–504. IEEE Computer Society, 2003.

[AYBDS06] Sihem Amer-Yahia, Chavdar Botev, Jochen Dörre, and Jayavel Shanmugasundaram. XQuery Full-Text extensions explained. *IBM Systems Journal*, 45(2):335–352, 2006.

[AYBS04] Sihem Amer-Yahia, Chavdar Botev, and Jayavel Shanmugasundaram. Texquery: a full-text search extension to xquery. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of*

the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004, pages 583–594. ACM, 2004.

[AYCD06] Sihem Amer-Yahia, Emiran Curtmola, and Alin Deutsch. Flexible and efficient XML search with complex full-text predicates. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 575–586. ACM, 2006.

[AYCS02] Sihem Amer-Yahia, SungRan Cho, and Divesh Srivastava. Tree Pattern Relaxation. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, volume 2287 of *Lecture Notes in Computer Science*, pages 496–513. Springer, 2002.

[AYKM+05] Sihem Amer-Yahia, Nick Koudas, Amélie Marian, Divesh Srivastava, and David Toman. Structure and Content Scoring for XML. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 361–372. ACM, 2005.

[AYL06] Sihem Amer-Yahia and Mounia Lalmas. XML search: languages, INEX and scoring. *SIGMOD Record*, 35(4):16–23, 2006.

[AYLP04] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Shashank Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 83–94. ACM, 2004.

[BAYS06] Chavdar Botev, Sihem Amer-Yahia, and Jayavel Shanmugasundaram. Expressiveness and Performance of Full-Text Search Languages. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*, pages 349–367. Springer, 2006.

[BCF+03] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery/, November 2003.

[Ber01] Michael K. Bergman. The Deep Web: Surfacing Hidden Value. *Journal of Electronic Publishing*, 7, aug 2001.

[BG02] Jan-Marco Bremer and Michael Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. In Mary F. Fernandez and Yannis Papakonstantinou, editors, *Proceedings of the Fifth International Workshop on the Web and Databases, WebDB 2002, Madison, Wisconsin, USA, June 6-7, 2002, in conjunction with ACM PODS/SIGMOD 2002. Informal proceedings*, pages 1–6, 2002.

[BG03] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/, dec 2003.

[BG06] Jan-Marco Bremer and Michael Gertz. Integrating document and data retrieval based on XML. *VLDB Journal*, 15(1):53–83, 2006.

[BGL+99] Chaitanya K. Baru, Amarnath Gupta, Bertram Ludäscher, Richard Marciano, Yannis Papakonstantinou, Pavel Velikhov, and Vincent Chu. XML-Based Information Mediation with MIX. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 597–599. ACM Press, 1999.

[BGTC09] Nikos Bikakis, Nektarios Gioldasis, Chrisa Tsinaraki, and Stavros Christodoulakis. Semantic Based Access over XML Data. In *Proceedings of the 2nd World Summit on the Knowledge Society: Visioning and Engineering the Knowledge Society. A Web Science Perspective, WSKS '09, Chania, Crete, Greece*, pages 259–267, Berlin, Heidelberg, 2009. Springer-Verlag.

[BHK+03] Andrey Balmin, Vagelis Hristidis, Nick Koudas, Yannis Papakonstantinou, Divesh Srivastava, and Tianqiu Wang. A System for Keyword Proximity Search on XML Databases. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 1069–1072. Morgan Kaufmann, 2003.

[BKLW99] Susanne Busse, Ralf-Detlef Kutsche, Ulf Leser, and Herbert Weber. Federated Information Systems: Concepts, Terminology and Architectures. Technical Report 99-9, Technische Universität Berlin, Fachbereich 13 Informatik, 1999.

[BLN86] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computer Surveys*, 18(4):323–364, 1986.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks / Proceedings of WWW7*, 30(1-7):107–117, April 1998.

[BRDN10] Akanksha Baid, Ian Rae, AnHai Doan, and Jeffrey F. Naughton. Toward industrial-strength keyword search systems over relational data. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 717–720. IEEE, 2010.

[BRL+10] Akanksha Baid, Ian Rae, Jiexing Li, AnHai Doan, and Jeffrey F. Naughton. Toward Scalable Keyword Search over Relational Data. *PVLDB*, 3(1):140–149, 2010.

[BS05] Chavdar Botev and Jayavel Shanmugasundaram. Context-Sensitive Keyword Search and Ranking for XML. In AnHai Doan, Frank Neven, Robert McCann, and Geert Jan Bex, editors, *Proceedings of the Eight International Workshop on the Web & Databases (WebDB 2005), Baltimore, Maryland, USA, Collocated mith ACM SIGMOD/PODS 2005, June 16-17, 2005*, pages 115–120, 2005.

[BSAY04] Chavdar Botev, Jayavel Shanmugasundaram, and Sihem Amer-Yahia. A TeXQuery-Based XML Full-Text Search Engine. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 943–944. ACM, 2004.

[BvHH+03] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/, dec 2003.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[Cal00] James P. Callan. Distributed Information Retrieval. In W.B. Croft, editor, *Advances in Information Retrieval*, chapter 5, pages 127–150. luwer Academic Publishers, 2000.

[CBC+09] Eric Chu, Akanksha Baid, Xiaoyong Chai, AnHai Doan, and Jeffrey F. Naughton. Combining keyword search and forms for ad hoc querying of databases. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 349–360. ACM, 2009.

[CC01] James P. Callan and M. Connell. Query-based sampling of text databases. *ACM Trans. Inf. Syst.*, 19(2):97–130, 2001.

[CCD99] James P. Callan, Margaret E. Connell, and Aiqun Du. Automatic Discovery of Language Models for Text Databases. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 479–490. ACM Press, 1999.

[CdSV+02] Pável Calado, Altigran Soares da Silva, Rodrigo C. Vieira, Alberto H. F. Laender, and Berthier A. Ribeiro-Neto. Searching web databases by structuring keyword-based queries. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 4-9, 2002*, pages 26–33. ACM, 2002.

[CEL+02] David Carmel, Nadav Efraty, Gad M. Landau, Yoëlle S. Maarek, and Yosi Mass. An Extension of the Vector Space Model for Querying XML Documents via XML Fragements. In *ACM SIGIR'2002 Workshop on XML and IR*, Tampere, Finland, 2002.

[CGL+10] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati, and Marco Ruzzi. Using OWL in Data Integration. In Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Semantic Web Information Management*, pages 397–424. Springer, 2010.

[CHL+04] Kevin Chen-Chuan Chang, Bin He, Chengkai Li, Mitesh Patel, and Zhen Zhang. Structured Databases on the Web: Observations and Implications. *SIGMOD Record*, 33(3):61–70, 2004.

[CHS+95] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, Myron Flickner, Allen Luniewski, Wayne Niblack, Dragutin Petkovic, Joachim Thomas II, John H. Williams, and Edward L. Wimmers. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *RIDE-DOM*, pages 124–131, 1995.

[CHZ05] Kevin Chen-Chuan Chang, Bin He, and Zhen Zhang. Toward Large Scale Integration: Building a MetaQuerier over Databases on the Web. In *CIDR*, pages 44–55, 2005.

[CK02] Taurai Tapiwa Chinenyanga and Nicholas Kushmerick. An expressive and efficient language for XML information retrieval. *JASIST*, 53(6):438–453, 2002.

[CKKS05] Sara Cohen, Yaron Kanza, Benny Kimelfeld, and Yehoshua Sagiv. Interconnection semantics for keyword search in XML. In Otthein Herzog, Hans-Jörg Schek, Norbert Fuhr, Abdur Chowdhury, and Wilfried Teiken, editors, *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pages 389–396. ACM, 2005.

[CKS01] Sara Cohen, Yaron Kanza, and Yehoshua Sagiv. SQL4X: A Flexible Query Language for XML and Relational Databases. In Giorgio Ghelli and Gösta Grahne, editors, *Database Programming Languages, 8th International Workshop, DBPL 2001, Frascati, Italy, September 8-10, 2001, Revised Papers*, volume 2397 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 2001.

[CLC95] James P. Callan, Zhihong Lu, and W. Bruce Croft. Searching Distributed Collections with Inference Networks. In Edward A. Fox, Peter Ingwersen, and Raya Fidel, editors, *SIGIR'95, Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Seattle, Washington, USA, July 9-13, 1995 (Special Issue of the SIGIR Forum)*, pages 21–28. ACM Press, 1995.

[CM08] Andrea Calì and Davide Martinenghi. Querying Data under Access Limitations. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 50–59. IEEE, 2008.

[CMKS03] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. XSEarch: A Semantic Search Engine for XML. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 45–56. Morgan Kaufmann, 2003.

[CMM+03] David Carmel, Yoëlle S. Maarek, Matan Mandelbrod, Yosi Mass, and Aya Soffer. Searching XML documents via XML fragments. In *SIGIR 2003: Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, July 28 - August 1, 2003, Toronto, Canada*, pages 151–158. ACM, 2003.

[Con97] Stefan Conrad. *Föderierte Datenbanksysteme: Konzepte der Datenintegration*. Springer-Verlag, Berlin/Heidelberg, 1997.

[COZ07] Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. ACM, 2007.

[CRS99] Boris Chidlovskii, Claudia Roncancio, and Marie-Luise Schneider. Semantic Cache Mechanism for Heterogeneous Web Querying. *Computer Networks*, 31(11-16):1347–1360, 1999.

[CRW05] Surajit Chaudhuri, Raghu Ramakrishnan, and Gerhard Weikum. Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In *CIDR*, pages 1–12, 2005.

[CWLL09] Yi Chen, Wei Wang, Ziyang Liu, and Xuemin Lin. Keyword search on structured and semi-structured data. In Ugur Çetintemel, Stanley B.

Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 1005–1010. ACM, 2009.

[DD99] Ruxandra Domenig and Klaus R. Dittrich. An Overview and Classification of Mediated Query Systems. *SIGMOD Record*, 28(3):63–72, 1999.

[DD00] Ruxandra Domenig and Klaus R. Dittrich. A Query based Approach for Integrating Heterogeneous Data Sources. In *Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 6-11, 2000*, pages 453–460. ACM, 2000.

[DD01a] Ruxandra Domenig and Klaus R. Dittrich. Query preprocessing for integrated search in heterogeneous data sources. In Andreas Heuer, Frank Leymann, and Denny Priebe, editors, *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), 9. GI-Fachtagung, Oldenburg, 7.-9. März 2001, Proceedings*, Informatik Aktuell, pages 154–163. Springer, 2001.

[DD01b] Ruxandra Domenig and Klaus R. Dittrich. SINGAPORE: A system for querying heterogeneous data sources. In *ICDE 2001, Demo Session Abstracts (Informal Proceedings)*, pages 10–11, 2001.

[Dec04] Torsten Declercq. Stichwortsuche in heterogenen, semi-strukturierten Datenbeständen (in german). Master's thesis, Otto-von-Guericke-Universität Magdeburg, January 2004.

[DEGP98a] Shaul Dar, Gadi Entin, Shai Geva, and Eran Palmon. DTL's DataSpot: database exploration as easy as browsing the Web. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 590–592. ACM Press, 1998.

[DEGP98b] Shaul Dar, Gadi Entin, Shai Geva, and Eran Palmon. DTL's DataSpot: Database Exploration Using Plain Language. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 645–649. Morgan Kaufmann Publishers Inc., 1998.

[Dew04] Melvil Dewey. Dewey Decimal Classification. Internet, etext,, jun 2004. http://www.gutenberg.org/etext/12513 (retrieved 2008-07-11).

[DFJ+96] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. Semantic Data Caching and Replacement. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 330–341. Morgan Kaufmann, 1996.

[DFKR99] Hasan Davulcu, Juliana Freire, Michael Kifer, and I. V. Ramakrishnan. A Layered Architecture for Querying Dynamic Web Content. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD*

1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA., pages 491–502. ACM Press, 1999.

[DG97] Oliver M. Duschka and Michael R. Genesereth. Query Planning in Infomaster. In *SAC*, pages 109–111, 1997.

[DH05] AnHai Doan and Alon Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26(1):83–94, 2005.

[DH07] Xin Dong and Alon Y. Halevy. Indexing dataspaces. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 43–54. ACM, 2007.

[DRR+03] Claude Delobel, Chantal Reynaud, Marie-Christine Rousset, Jean-Pierre Sirot, and Dan Vodislav. Semantic integration in Xyleme: a uniform tree-based approach. *Data & Knowledge Engineering*, 44(3):267–298, 2003.

[DZN10] Elena Demidova, Xua Zhou, and Wolfgang Nejdl. IQ^p: Incremental Query Construction, a Probabilistic Approach. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 349–352. IEEE, 2010.

[DZZN09] Elena Demidova, Xuan Zhou, Gideon Zenz, and Wolfgang Nejdl. SUITS: Faceted User Interface for Constructing Structured Queries from Keywords. In Xiaofang Zhou, Haruo Yokota, Ke Deng, and Qing Liu, editors, *DASFAA*, volume 5463 of *Lecture Notes in Computer Science*, pages 772–775. Springer, 2009.

[EBG+07] Robert Ennals, Eric A. Brewer, Minos N. Garofalakis, Michael Shadle, and Prashant Gandhi. Intel mash maker: join the web. *SIGMOD Record*, 36(4):27–33, 2007.

[FAB+02] Irini Fundulaki, Bernd Amann, Catriel Beeri, Michel Scholl, and Anne-Marie Vercoustre. ST_YX: Connecting the XML Web to the World of Semantics. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, volume 2287 of *Lecture Notes in Computer Science*, pages 759–761. Springer, 2002.

[Feg04] Leonidas Fegaras. XQuery Processing with Relevance Ranking. In Zohra Bellahsene, Tova Milo, Michael Rys, Dan Suciu, and Rainer Unland, editors, *Database and XML Technologies, Second International XML Database Symposium, XSym 2004, Toronto, Canada, August 29-30, 2004, Proceedings*, volume 3186 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2004.

[FG01] Norbert Fuhr and Kai Großjohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In W. Bruce Croft, David J. Harper, Donald H. Kraft, and Justin Zobel, editors, *SIGIR 2001: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, September 9-13, 2001, New Orleans, Louisiana, USA*, pages 172–180. ACM, 2001.

[FG04] Norbert Fuhr and Kai Großjohann. XIRQL: An XML Query Language based on Information Retrieval Concepts. *ACM Trans. Inf. Syst.*, 22(2):313–356, 2004.

[FHM05] Michael Franklin, Alon Halevy, and David Maier. From Databases to Dataspaces: A New Abstraction for Information Management. *SIGMOD Record*, 34(4):27–33, December 2005.

[FKM00] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating Keyword Search into XML Query Processing. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, Computer Networks, pages 119–135, June 2000.

[FLM99] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational Plans for Data Integration. In *Proceedings of the IJCAI-99 Workshop on Intelligent Information Integration, Held on July 31, 1999 in conjunction with the Sixteenth International Joint Conference on Artificial Intelligence City Conference Center, Stockholm, Sweden*, volume 23 of *CEUR Workshop Proceedings*, 1999.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal Aggregation Algorithms for Middleware. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, pages 102–113. ACM, 2001.

[Fuh92] Norbert Fuhr. Probabilistic Models in Information Retrieval. *Computer Journal*, 35(3):243–255, 1992.

[GBMS99] Cheng Hian Goh, Stéphane Bressan, Stuart E. Madnick, and Michael Siegel. Context Interchange: New Features and Formalisms for the Intelligent Integration of Information. *ACM Transactions on Information Systems (TOIS)*, 17(3):270–293, 1999.

[GCGMP97] Luis Gravano, Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. STARTS: Stanford Proposal for Internet Meta-Searching (Experience Paper). In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 207–218. ACM Press, 1997.

[GDSS03] Ingolf Geist, Torsten Declercq, Kai-Uwe Sattler, and Eike Schallehn. Query Reformulation for Keyword Searching in Mediator Systems. Technical Report 8, Fakultät für Informatik, Universität Magdeburg, 2003.

[Gei04] Ingolf Geist. Index-based keyword search in mediator systems. In Wolfgang Lindner, Marco Mesiti, Can Türker, Yannis Tzitzikas, and Athena Vakali, editors, *Current Trends in Database Technology - EDBT 2004 Workshops, EDBT 2004 Workshops PhD, DataX, PIM, P2P&DB, and ClustWeb, Heraklion, Crete, Greece, March 14-18, 2004, Revised Selected Papers*, volume 3268 of *Lecture Notes in Computer Science*, pages 24–33. Springer, 2004.

[GGM95] Luis Gravano and Hector Garcia-Molina. Generalizing GlOSS to Vector-Space Databases and Broker Hierarchies. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 78–89. Morgan Kaufmann, 1995.

[GGMT94] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The Effectiveness of GlOSS for the Text Database Discovery Problem. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 126–137. ACM Press, 1994.

[GIG01] Noah Green, Panagiotis G. Ipeirotis, and Luis Gravano. SDLIP + STARTS = SDARTS a protocol and toolkit for metasearching. In *Proceedings of the first ACM/IEEE-CS joint conference on Digital libraries*, pages 207–214. ACM Press, 2001.

[GMPQ+97] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems (JIIS)*, 8(2):117–132, 1997.

[Gru91] Thomas R. Gruber. The Role of Common Ontology in Achieving Sharable, Reusable Knowledge Bases. In *KR*, pages 601–602, 1991.

[GSBS03] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the*

2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, pages 16–27. ACM, 2003.

[GSVGM98] Roy Goldman, Narayanan Shivakumar, Suresh Venkatasubramanian, and Hector Garcia-Molina. Proximity Search in Databases. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 26–37. Morgan Kaufmann Publishers Inc., 1998.

[GSW96] Sha Guo, Wei Sun, and Mark Allen Weiss. On Satisfiability, Equivalence, and Impication Problems Involving Conjunctive Queries in Database Systems. *IEEE Trans. Knowl. Data Eng.*, 8(4):604–616, 1996.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 436–445. Morgan Kaufmann, 1997.

[GW00] Roy Goldman and Jennifer Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 285–296. ACM, 2000.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[HBEV04] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF Query Languages. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, volume 3298 of *Lecture Notes in Computer Science*, pages 502–517. Springer, 2004.

[HBN+01] Arvind Hulgeri, Gaurav Bhalotia, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarsha. Keyword Search in Databases. *IEEE Data Eng. Bull.*, 24(3):22–32, 2001.

[HFM06] Alon Halevy, Michael Franklin, and David Maier. Principles of Dataspace Systems. In *PODS 2006*, pages 1–9. ACM, 2006.

[HGP03] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 850–861. Morgan Kaufmann, 2003.

[HHMW07] Theo Härder, Michael Peter Haustein, Christian Mathis, and Markus Wagner. Node labeling schemes for dynamic XML documents reconsidered. *Data & Knowledge Engineering*, 60(1):126–149, 2007.

[HKPS06] Vagelis Hristidis, Nick Koudas, Yannis Papakonstantinou, and Divesh Srivastava. Keyword Proximity Search in XML Trees. *IEEE Trans. Knowl. Data Eng.*, 18(4):525–539, 2006.

[HKWY97] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing Queries Across Diverse Data Sources. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 276–285. Morgan Kaufmann, 1997.

[HM09a] Thomas Hornung and Wolfgang May. Deep Web Queries in a Semantic Web Environment. In *Business Information Systems Workshops, BIS 2009 International Workshops, Pozan, Poland, April 27-29, 2009, Revised Papers*, volume 37 of *Lecture Notes in Business Information Processing*, pages 39–50. Springer, 2009.

[HM09b] Thomas Hornung and Wolfgang May. Semantic Annotations and Querying of Web Data Sources. In *OTM Conferences (1)*, volume 5870 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2009.

[HML09] Thomas Hornung, Wolfgang May, and Georg Laussen. Process Algebra-Based Query Workflows. In *Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June, 8-12, 2009, Proceedings*, volume 5565 of *Lecture Notes in Computer Science*, pages 440–454. Springer, 2009.

[HMYW03] Hai He, Weiyi Meng, Clement T. Yu, and Zonghuan Wu. WISE-Integrator: An Automatic Integrator of Web Search Interfaces for E-Commerce. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 357–368. Morgan Kaufmann, 2003.

[HMYW05] Hai He, Weiyi Meng, Clement T. Yu, and Zonghuan Wu. WISE-Integrator: A System for Extracting and Integrating Complex Web Search Interfaces of the Deep Web. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1314–1317. ACM, 2005.

[HN02] Arvind Hulgeri and Charuta Nakhe. Keyword Searching and Browsing in Databases using BANKS. In *Proceedings of the 18th International*

Conference on Data Engineering (ICDE'02), page 431. IEEE Computer Society, 2002.

[Höp05] Hagen Höpfner. *Relevanz von Änderungen für Datenbestände mobiler Clients*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, Magdeburg, 2005.

[HP02] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword Search in Relational Databases. In *VLDB'2002*, pages 670–681. VLDB Endowment, 2002.

[HP04] Vagelis Hristidis and Yannis Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB Journal*, 13(1):49–70, 2004.

[HPB03] Vagelis Hristidis, Yannis Papakonstantinou, and Andrey Balmin. Keyword Proximity Search on XML Graphs. In *19th International Conference on Data Engineering,March 05 - 08, 2003, Bangalore, India*, pages 367–378. IEEE Computer Society, 2003.

[HRO06] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: the teenage years. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 9–16. VLDB Endowment, 2006.

[Hul97] Richard Hull. Managing Semantic Heterogeneity in Databases: A Theoretical Perspective. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 51–61. ACM Press, 1997.

[HWYY07] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. BLINKS: Ranked Keyword Searches on Graphs. In Chan et al. [COZ07], pages 305–316.

[IBG02] Panagiotis G. Ipeirotis, Tom Barry, and Luis Gravano. Extending SDARTS: extracting metadata from web databases and interfacing with the open archives initiative. In *Proceedings of the second ACM/IEEE-CS joint conference on Digital libraries*, pages 162–170. ACM Press, 2002.

[IG02] P.G. Ipeirotis and L. Gravano. Distributed Search over the Hidden Web: Hierarchical Database Sampling and Selection. In *VLDB 2002*, pages 394–405, 2002.

[IKNG09] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. An Architecture for Recycling Intermediates in a Column-Store. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 309–320. ACM, 2009.

[IKNG10] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. An Architecture for Recycling Intermediates in a Column-Store. *ACM Trans. Database Syst.*, 35(4):24, 2010.

[Inm96] W. H. Inmon. The Data Warehouse and Data Mining. *Communications of the ACM (CACM)*, 39(11):49–50, 1996.

[JBB+97] Roberto J. Bayardo Jr., William Bohrer, Richard S. Brice, Andrzej Cichocki, Jerry Fowler, Abdelsalam Helal, Vipul Kashyap, Tomasz Ksiezyk, Gale Martin, Marian H. Nodine, Mosfeq Rashid, Marek Rusinkiewicz, Ray Shea, C. Unnikrishnan, Amy Unruh, and Darrell Woelk. InfoSleuth: Semantic Integration of Information in Open and Dynamic Environments (Experience Paper). In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 195–206. ACM Press, 1997.

[JCE+07] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In Chan et al. [COZ07], pages 13–24.

[Kar03] Marcel Karnstedt. Semantisches Caching in ontologiebasierten Mediatoren. Master's thesis, Martin-Luther-Universität Halle-Wittenberg, Fachbereich Mathematik und Informatik,Institut für Informatik, D–06120 HALLE (Saale), October 2003.

[KB96] Arthur M. Keller and Julie Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB J.*, 5(1):35–47, 1996.

[Ken91] William Kent. Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings.*, pages 147–160. Morgan Kaufmann, 1991.

[KLK91] Ravi Krishnamurthy, Witold Litwin, and William Kent. Language Features for Interoperability of Databases with Schematic Discrepancies. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991.*, pages 40–49. ACM Press, 1991.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.

[KMA+01] Craig A. Knoblock, Steven Minton, José Luis Ambite, Naveen Ashish, Ion Muslea, Andrew Philpot, and Sheila Tejada. The Ariadne Approach

to Web-Based Information Integration. *Int. J. Cooperative Inf. Syst.*, 10(1-2):145–169, 2001.

[KNS02] Yaron Kanza, Werner Nutt, and Yehoshua Sagiv. Querying Incomplete Information in Semistructured Data. *J. Comput. Syst. Sci.*, 64(3):655–693, 2002.

[Kos00] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[KPC+05] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional Expansion For Keyword Search on Graph Databases. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, pages 505–516. ACM, 2005.

[KS91] Won Kim and Jungyun Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*, 24(12):12–18, 1991.

[KS00] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, 2000.

[KS01] Yaron Kanza and Yehoshua Sagiv. Flexible Queries Over Semistructured Data. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. ACM, 2001.

[KSGH03] Marcel Karnstedt, Kai-Uwe Sattler, Ingolf Geist, and Hagen Höpfner. Semantic Caching in Ontology-based Mediator Systems. In Robert Tolksdorf and Rainer Eckstein, editors, *Berliner XML Tage 2003, 13.-15. Oktober 2003 in Berlin*, pages 155–169. XML-Clearinghouse, 2003.

[Lan08] Andreas Langegger. Virtual data integration on the web: novel methods for accessing heterogeneous and distributed data with rich semantics. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services, iiWAS '08*, pages 559–562, New York, NY, USA, 2008. ACM.

[LC99] Dongwon Lee and Wesley W. Chu. Semantic Caching via Query Matching for Web Sources. In *Proceedings of the 1999 ACM CIKM International Conference on Information and Knowledge Management, Kansas City, Missouri, USA, November 2-6, 1999*, pages 77–85. ACM, 1999.

[LC01] Dongwon Lee and Wesley W. Chu. Towards Intelligent Semantic Caching for Web Sources. *J. Intell. Inf. Syst.*, 17(1):23–45, 2001.

[LCY+07] Yunyao Li, Ishan Chaudhuri, Huahai Yang, Satinder Singh, and H. V. Jagadish. DaNaLIX: a domain-adaptive natural language interface for

querying XML. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1165–1168. ACM, 2007.

[Len02] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In Lucian Popa, editor, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246. ACM, 2002.

[Lev66] Vladimir Levenshtein. Binary codes of correcting deletions, insertions, and reversals. *Soviet Physics - Doklady 10*, 10:707 – 710, 1966.

[LF04] Patrick Lehti and Peter Fankhauser. XML Data Integration with OWL: Experiences and Challenges. In *2004 Symposium on Applications and the Internet (SAINT 2004), 26-30 January 2004, Tokyo, Japan*, pages 160–170. IEEE Computer Society, 2004.

[LFO+11] Guoliang Li, Jianhua Feng, Beng Chin Ooi, Jianyong Wang, and Lizhu Zhou. An effective 3-in-1 keyword search method over heterogeneous data sources. *Inf. Syst.*, 36(2):248–266, 2011.

[LFZ08] Guoliang Li, Jianhua Feng, and Lizhu Zhou. Retune: Retrieving and Materializing Tuple Units for Effective Keyword Search over Relational Databases. In Qing Li, Stefano Spaccapietra, Eric S. K. Yu, and Antoni Olivé, editors, *ER*, volume 5231 of *Lecture Notes in Computer Science*, pages 469–483. Springer, 2008.

[LFZW11] Guoliang Li, Jianhua Feng, Xiaofang Zhou, and Jianyong Wang. Providing built-in keyword search capabilities in RDBMS. *VLDB J.*, 20(1):1–19, 2011.

[LGM01] Bertram Ludäscher, Amarnath Gupta, and Maryann E. Martone. Model-Based Mediation with Domain Maps. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 81–90. IEEE Computer Society, 2001.

[LLWZ07] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: top-k keyword query in relational databases. In Chan et al. [COZ07], pages 115–126.

[LLY+05] Fang Liu, Shuang Liu, Clement T. Yu, Weiyi Meng, Ophir Frieder, and David A. Grossman. Database selection in intranet mediators for natural language queries. In Otthein Herzog, Hans-Jörg Schek, Norbert Fuhr, Abdur Chowdhury, and Wilfried Teiken, editors, *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pages 229–230. ACM, 2005.

[LMM07] Xian Li, Weiyi Meng, and Xiaofeng Meng. EasyQuerier: A Keyword Based Interface for Web Database Integration System. In Kotagiri Ramamohanarao, P. Radha Krishna, Mukesh K. Mohania, and Ekawit Nantajeewarawat, editors, *Advances in Databases: Concepts, Systems and Applications, 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007, Proceedings*, volume 4443 of *Lecture Notes in Computer Science*, pages 936–942. Springer, 2007.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering Queries Using Views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 95–104. ACM Press, 1995.

[LN01] Qiong Luo and Jeffrey F. Naughton. Form-Based Proxy Caching for Database-Backed Web Sites. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 191–200. Morgan Kaufmann, 2001.

[LOF+08] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In Wang [Wan08], pages 903–914.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufmann, 1996.

[LSS96] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-Database Systems. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 239–250. Morgan Kaufmann, 1996.

[Luo06] Gang Luo. Efficient Detection of Empty-Result Queries. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 1015–1025. ACM, 2006.

[Luo09] Yi Luo. *SPARK: A Keyword Search System on Relational Databases*. PhD thesis, The School of Computer Science and Engineering, The University of New South Wales., 2009.

[LWB08] Andreas Langegger, Wolfram Wöß, and Martin Blöchl. A Semantic Web Middleware for Virtual Data Integration on the Web. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 493–507. Springer Berlin / Heidelberg, 2008.

[LYJ04] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-Free XQuery. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 72–83. Morgan Kaufmann, 2004.

[LYJ07] Yunyao Li, Huahai Yang, and H. V. Jagadish. NaLIX: A generic natural language search environment for XML data. *ACM Trans. Database Syst.*, 32(4), 2007.

[LYJ08] Yunyao Li, Cong Yu, and H. V. Jagadish. Enabling Schema-Free XQuery with meaningful query focus. *VLDB J.*, accepted for publication:355–377, 2008.

[LYMC06] Fang Liu, Clement T. Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 563–574. ACM, 2006.

[May05] Wolfgang May. Logic-based XML data integration: a semi-materialized approach. *Jornal of Applied Logic*, 3(1):271–307, 2005.

[MBG04] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[MBHW02] Holger Meyer, Ilvio Bruder, Andreas Heuer, and Gunnar Weber. The Xircus Search Engine. In Norbert Fuhr, Norbert Gövert, Gabriella Kazai, and Mounia Lalmas, editors, *Proceedings of the First Workshop of the INitiative for the Evaluation of XML Retrieval (INEX), Schloss Dagstuhl, Germany, December 9-11, 2002*, pages 119–124, 2002.

[MP00] Kevin D. Munroe and Yannis Papakonstantinou. BBQ: A Visual Interface for Integrated Browsing and Querying of XML. In Hiroshi Arisawa and Tiziana Catarci, editors, *Advances in Visual Information Management, proceedings of the Fifth Working Conference on Visual Database*

Systems (VDB5), Fukuoka, Japan, May 10-12, 2000, volume 168 of *IFIP Conference Proceedings*, pages 277–296. Kluwer, 2000.

[MS02] Jim Melton and Alan R. Simon. *SQL 1999*. Morgan Kaufmann Publishers, 2002.

[MV00] Ute Masermann and Gottfried Vossen. Design and Implementation of a Novel Approach to Keyword Searching in Relational Databases. In Julius Stuller, Jaroslav Pokorný, Bernhard Thalheim, and Yoshifumi Masunaga, editors, *Current Issues in Databases and Information Systems, East-European Conference on Advances in Databases and Information Systems Held Jointly with International Conference on Database Systems for Advanced Applications, ADBIS-DASFAA 2000, Prague, Czech Republic, September 5-8, 2000, Proceedings*, volume 1884 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2000.

[MYL02] Weiyi Meng, Clement T. Yu, and King-Lup Liu. Building efficient and effective metasearch engines. *ACM Computing Surveys*, 34(1):48–89, 2002.

[MYP09] Alexander Markowetz, Yin Yang, and Dimitris Papadias. Keyword Search over Relational Tables and Streams. *ACM Trans. Database Syst.*, 34(3):17:1–17:51, 2009.

[NLF99] Felix Naumann, Ulf Leser, and Johann Christoph Freytag. Quality-driven Integration of Heterogenous Information Systems. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 447–458. Morgan Kaufmann, 1999.

[Noy04] Natasha F. Noy. Semantic integration: a survey of ontology-based approaches. *ACM Sigmod Record*, 33(4):65–70, 2004.

[OOP+04] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 903–908. ACM, 2004.

[ÖV11] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, New York Dordrecht Heidelberg London, 3rd edition, 2011.

[PBMW98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Computer Science Department, Stanford University, January 1998.

[PgL11] Jaehui Park and Sang goo Lee. Keyword search in relational databases. *Knowl. Inf. Syst.*, 26(2):175–193, 2011.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 251–260. IEEE Computer Society, 1995.

[PH01] Rachel Pottinger and Alon Y. Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2-3):182–198, 2001.

[Ple81] Jan Plesnik. A bound for the Steiner tree problem in graphs. *Math. Slovaca*, (31):155–163, 1981.

[QYC09] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Keyword search in databases: the power of RDBMS. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 681–694. ACM, 2009.

[QYC11] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Scalable keyword search on large data streams. *VLDB J.*, 20(1):35–57, 2011.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal The International Journal on Very Large Data Bases*, 10(4):334 – 350, dec 2001.

[RR99] Sudha Ram and V. Ramesh. Schema Integration: Past, Present, and Future. In A. K. Elmagarmid, A. Sheth, and M. Rusinkiewicz, editors, *Management of Heterogeneous and Autonomous Database Systems*, pages 119 – 155. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[SBJ+01] M. Saelee, Steven M. Beitzel, Eric C. Jensen, David A. Grossman, and Ophir Frieder. Intranet Mediators: A Prototype. In *2001 International Symposium on Information Technology (ITCC 2001), 2-4 April 2001, Las Vegas, NV, USA*, pages 389–394. IEEE Computer Society, 2001.

[SC03] Luo Si and Jamie Callan. Relevant document distribution estimation method for resource selection. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 298–305. ACM Press, 2003.

[SCH+97] Munindar P. Singh, Philip Cannata, Michael N. Huhns, Nigel Jacobs, Tomasz Ksiezyk, KayLiang Ong, Amit P. Sheth, Christine Tomlinson, and Darrell Woelk. The Carnot Heterogeneous Database Project: Implemented Applications. *Distributed and Parallel Databases*, 5(2):207–225, 1997.

[Sch98] Ingo Schmitt. *Schemaintegration für den Entwurf Föderierter Datenbanken*, volume 43 of *Dissertationen zu Datenbanken und Informationssystemen*. infix-Verlag, Sankt Augustin, 1998.

[Sch02] Torsten Schlieder. Schema-Driven Evaluation of Approximate Tree-Pattern Queries. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, volume 2287 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2002.

[Sch04] Eike Schallehn. *Efficient Similarity-based Operations for Data Integration*. PhD thesis, Fakultät für Informatik, Universität Magdeburg, mar 2004.

[SCS03] Kai-Uwe Sattler, Stefan Conrad, and Gunter Saake. Interactive example-driven integration and reconciliation for accessing database federations. *Inf. Syst.*, 28(5):393–414, 2003.

[SDK+07] Marcos Antonio Vaz Salles, Jens-Peter Dittrich, Shant Kirakos Karakashian, Olivier René Girard, and Lukas Blunschi. iTrails: Pay-as-you-go Information Integration in Dataspaces. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 663–674. ACM, 2007.

[Sel88a] Timos K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Inf. Syst.*, 13(2):175–185, 1988.

[Sel88b] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13:23–52, March 1988.

[Sem05] Semantic web. http://www.w3.org/2001/sw/, jul 2005.

[SGB+07] Feng Shao, Lin Guo, Chavdar Botev, Anand Bhaskar, Muthiah M. Muthiah Chettiar, Fan Yang, and Jayavel Shanmugasundaram. Efficient Keyword Search over Virtual XML Views. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti,

Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1057–1068. ACM, 2007.

[SGHS03] Kai-Uwe Sattler, Ingolf Geist, Rainer Habrecht, and Eike Schallehn. Konzeptbasierte Anfrageverarbeitung in Mediatorsystemen. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, 26.-28. Februar 2003, Leipzig*, volume 26 of *LNI*, pages 78–97. GI, 2003.

[SGS04] Eike Schallehn, Ingolf Geist, and Kai-Uwe Sattler. Supporting Similarity Operations Based on Approximate String Matching on the Web. In Robert Meersman and Zahir Tari, editors, *CoopIS/DOA/ODBASE (1)*, volume 3290 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2004.

[SGS05] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. Concept-based querying in mediator systems. *The VLDB Journal*, 14(1):97–111, 2005.

[SH07] Felipe Victolla Silveira and Carlos A. Heuser. A Two Layered Approach for Querying Integrated XML Sources. In *Proceedings of the 11th International Database Engineering and Applications Symposium*, pages 3–11, Washington, DC, USA, 2007. IEEE Computer Society.

[She99] Amit Sheth. Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics. In M. F. Goodchild, M. J. Egenhofer, R. Fegeas, and C. A. Kottman, editors, *Interoperating Geographic Information Systems*, pages 5 – 30. Academic Publishers, 1999.

[SHK05] Stefan Seltzsam, Roland Holzhauser, and Alfons Kemper. Semantic Caching for Web Services. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings*, volume 3826 of *Lecture Notes in Computer Science*, pages 324–340. Springer, 2005.

[Sin01] Amit Singhal. Modern Information Retrieval: A Brief Overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[SLDG07] Mayssam Sayyadian, Hieu LeKhac, AnHai Doan, and Luis Gravano. Efficient Keyword Search Across Heterogeneous Relational Databases. In *Proceedings of the 23rd International Conference on Data Engineering,*

ICDE 2007, April 15-20, 2007, The Marmara Hotel, Istanbul, Turkey, pages 346–355. IEEE, 2007.

[SM83] G. Salton and M.J. McGill. *An Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[SPD92] Stefano Spaccapietra, Christine Parent, and Yann Dupont. Model Independent Assertions for Integration of Heterogeneous Schemas. *The VLDB Journal*, 1(1):81–126, 1992.

[SSH05] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken: Implementierungstechniken*. mitp-Verlag, Bonn, 2nd edition, 2005.

[SSV96] Peter Scheuermann, Junho Shim, and Radek Vingralek. WATCHMAN : A Data Warehouse Intelligent Cache Manager. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 51–62. Morgan Kaufmann, 1996.

[STW01] Sergej Sizov, Anja Theobald, and Gerhard Weikum. Ähnlichkeitssuche auf XML-Daten. *Datenbank-Spektrum*, 1:59–67, 2001.

[SW03] Qi Su and Jennifer Widom. Indexing Relational Database Content Offline for Efficient Keyword-Based Search. Technical report, Stanford University, Database Group, February 2003. http://dbpubs.stanford.edu/pub/2003-13.

[SW05] Qi Su and Jennifer Widom. Indexing Relational Database Content Offline for Efficient Keyword-Based Search. In *Proceedings of the Ninth International Database Engineering and Applications Symposium (IDEAS '05)*, jul 2005.

[TL08] Sandeep Tata and Guy M. Lohman. SQAK: doing more with keywords. In Wang [Wan08], pages 889–902.

[TRV98] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.

[TS05] Andrew Trotman and Börkur Sigurbjörnsson. NEXI, Now and Next. In Norbert Fuhr, Mounia Lalmas, Saadia Malik, and Zoltán Szlávik, editors, *Advances in XML Information Retrieval, Third International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2004, Dagstuhl Castle, Germany, December 6-8, 2004, Revised Selected Papers*, volume 3493 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 2005.

[TW01] Anja Theobald and Gerhard Weikum. Adding Relevance to XML. In Dan Suciu and Gottfried Vossen, editors, *The World Wide Web and Databases, Third International Workshop WebDB 2000, Dallas, Texas, USA, Maaay 18-19, 2000, Selected Papers*, volume 1997 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2001.

[TW02a] Anja Theobald and Gerhard Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, volume 2287 of *Lecture Notes in Computer Science*, pages 477–495. Springer, 2002.

[TW02b] Anja Theobald and Gerhard Weikum. The XXL search engine: ranked retrieval of XML data using indexes and ontologies. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, page 615. ACM, 2002.

[Ull90] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies.* W. H. Freeman & Co., 1990.

[Ull97] Jeffrey D. Ullman. Information Integration Using Logical Views. In Foto N. Afrati and Phokion G. Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 1997.

[VCdS$^+$02] Rodrigo C. Vieira, Pável Calado, Altigran Soares da Silva, Alberto H. F. Laender, and Berthier A. Ribeiro-Neto. Structuring keyword-based queries for web databases. In *ACM/IEEE Joint Conference on Digital Libraries, JCDL 2002, Portland, Oregon, USA, June 14-18, 2002, Proceedings*, pages 94–95. ACM, 2002.

[Vit85] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[VOPT08] Quang Hieu Vu, Beng Chin Ooi, Dimitris Papadias, and Anthony K. H. Tung. A graph method for keyword-based selection of the top-K databases. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 915–926. ACM, 2008.

[WA10] Fan Wang and Gagan Agrawal. Query Reuse Based Query Planning for Searches over the Deep Web. In Pablo Garcia Bringas, Abdelkader

Hameurlain, and Gerald Quirchmayr, editors, *Database and Expert Systems Applications, 21th International Conference, DEXA 2010, Bilbao, Spain, August 30 - September 3, 2010, Proceedings, Part II*, volume 6262 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2010.

[WAJ08] Fan Wang, Gagan Agrawal, and Ruoming Jin. Query Planning for Searching Inter-dependent Deep-Web Databases. In Bertram Ludäscher and Nikos Mamoulis, editors, *Scientific and Statistical Database Management, 20th International Conference, SSDBM 2008, Hong Kong, China, July 9-11, 2008, Proceedings*, volume 5069 of *Lecture Notes in Computer Science*, pages 24–41. Springer, 2008.

[Wan08] Jason Tsong-Li Wang, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008.* ACM, 2008.

[Wie92] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, 1992.

[Wie93] Gio Wiederhold. Intelligent Integration of Information. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.*, pages 434–437. ACM Press, 1993.

[WVV+01] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-Based Integration of Information – A Survey of Existing Approaches. In *Proceedings of the IJCAI-01 Workshop on Ontologies and Information Sharing*, Seattle, USA, 2001.

[WZP+07] Shan Wang, Jun Zhang, Zhaohui Peng, Jiang Zhan, and Qiuyue Wang. Study on Efficiency and Effectiveness of KSORD. In Guozhu Dong, Xuemin Lin, Wei Wang, Yun Yang, and Jeffrey Xu Yu, editors, *APWeb/WAIM*, volume 4505 of *Lecture Notes in Computer Science*, pages 6–17. Springer, 2007.

[XIG09] Yanwei Xu, Yoshiharu Ishikawa, and Jihong Guan. Effective Top-k Keyword Search in Relational Databases Considering Query Semantics. In Lei Chen, Chengfei Liu, Xiao Zhang, Shan Wang, Darijus Strasunskas, Stein L. Tomassen, Jinghai Rao, Wen-Syan Li, K. Selçuk Candan, Dickson K. W. Chiu, Yi Zhuang, Clarence A. Ellis, and Kwang-Hoon Kim, editors, *APWeb/WAIM Workshops*, volume 5731 of *Lecture Notes in Computer Science*, pages 172–184. Springer, 2009.

[XP05] Yu Xu and Yannis Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 537–538. ACM, 2005.

[XP08] Yu Xu and Yannis Papakonstantinou. Efficient LCA based keyword search in XML data. In Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu, editors, *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*, volume 261 of *ACM International Conference Proceeding Series*, pages 535–546. ACM, 2008.

[xpa07] XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, January 2007. W3C Recommendation.

[YJ06] Cong Yu and H. V. Jagadish. Schema Summarization. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 319–330. ACM, 2006.

[YJ07] Cong Yu and H. V. Jagadish. Querying Complex Structured Databases. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1010–1021. ACM, 2007.

[YLST07] Bei Yu, Guoliang Li, Karen R. Sollins, and Anthony K. H. Tung. Effective keyword-based selection of relational databases. In Chan et al. [COZ07], pages 139–150.

[YQC10] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. *Keyword Search in Databases.* Number Lecture 1 in Synthesis Lectures on Data Management. Morgan and Claypool Publishers, 2010. ISBN: 160845195X.

[ZHC05] Zhen Zhang, Bin He, and Kevin Chen-Chuan Chang. Light-weight Domain-based Form Assistant: Querying Web Databases On the Fly. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 97–108. ACM, 2005.

[Zlo75] Moshé M. Zloof. Query-by-Example: the Invocation and Definition of Tables and Forms. In Douglas S. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, pages 1–24. ACM, 1975.

[ZM98] Justin Zobel and Alistair Moffat. Exploring the Similarity Space. *SIGIR Forum*, 32(1):18–34, 1998.

[ZMR98] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.

[ZZDN08] Xuan Zhou, Gideon Zenz, Elena Demidova, and Wolfgang Nejdl. SUITS: Constructing Structured Queries from Keywords. Technical report, Forschungszentrum L3S, Universität Hannover, March 2008.

A. Data and Query Sets

A.1. Data Set

A.1.1. Concept schema IMDB

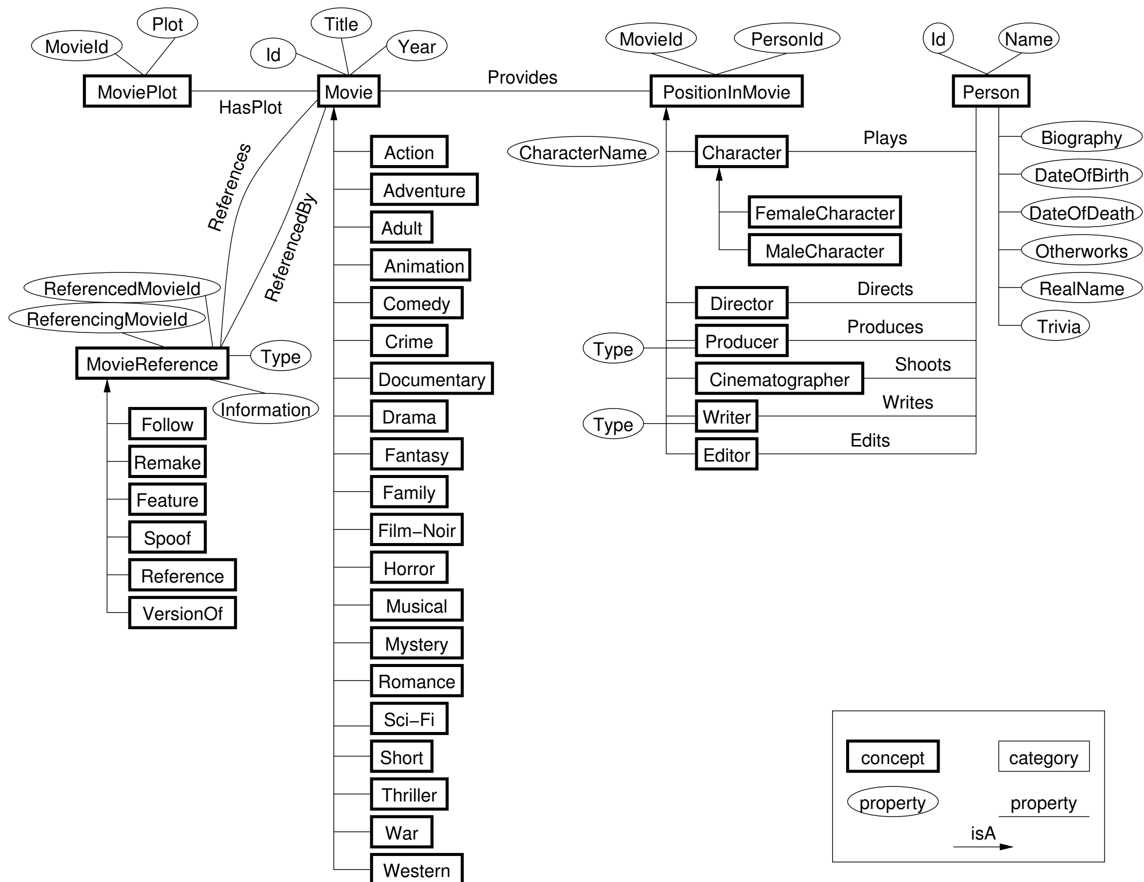


Figure A.1.: IMDB concept schema

A.2. Query Sets

A.2.1. Queries Cost Estimation

No.	**Query Set 1** **Query**
0	Person[Dateofbirth∼="posen"]
1	Person[Name∼="maida"]
2	Person[Name∼="melonie"]
3	Person[Name∼="essen"]
4	Person[Name∼="kadri"]
5	Person[Name∼="tirado"]
6	Person[Name∼="istvánné"]
7	Person[Name∼="reimann"]
8	Person[Name∼="bonneau"]
9	Person[Name∼="sobrino"]

No.	**Query set 2** **Query**
0	Person[Dateofbirth∼="paraná"]
1	Person[Name∼="harmon"]
2	Person[Name∼="izumi"]
3	Person[Dateofbirth∼="malmö"]
4	Person[Name∼="jenni"]
5	Person[Name∼="platt"]
6	Person[Name∼="draper"]
7	Person[Name∼="matej"]
8	Person[Name∼="barnaby"]
9	Person[Name∼="staley"]

No.	**Query set 3** **Query**
0	Person[Realname∼="wayne"]
1	Person[Otherworks∼="sony"]
2	Person[Otherworks∼="mamet"]
3	Person[Dateofbirth∼="norfolk"]
4	Person[Name∼="gonzalo"]
5	Person[Otherworks∼="beautiful"]
6	Person[Name∼="french"]
7	Person[Name∼="gemma"]
8	Person[Name∼="griffiths"]
9	Person[Otherworks∼="jake"]

No.	Query set 4 Query
0	Person[Name∼="cornell" ∧ Name∼="dupree"]
1	Person[Otherworks∼="flying" ∧ Otherworks∼="market"]
2	Person[Otherworks∼="delight" ∧ Otherworks∼="visual"]
3	Person[Biography∼="belonged" ∧ Biography∼="bite"]
4	Person[Biography∼="elaine" ∧ Biography∼="logo"]
5	Person[Name∼="farias" ∧ Name∼="henrique"]
6	Person[Dateofbirth∼="salvador" ∧ Realname∼="milton"]
7	Person[Name∼="galindo" ∧ Name∼="ofelia"]
8	Person[Dateofbirth∼="boulder" ∧ Dateofdeath∼="seattle"]
9	Person[Dateofbirth∼="arlington" ∧ Biography∼="paint"]

No.	Query set 5 Query
0	Person[Dateofbirth∼="manila" ∧ Biography∼="parade"]
1	Person[Realname∼="jacob" ∧ Realname∼="samuel"]
2	Person[Biography∼="exotic" ∧ Biography∼="remarkable"]
3	Person[Name∼="engel" ∧ Name∼="lilly"]
4	Person[Name∼="ginger" ∧ Name∼="mickey"]
5	Person[Otherworks∼="burger" ∧ Otherworks∼="joel"]
6	Person[Otherworks∼="development" ∧ Otherworks∼="devil"]
7	Person[Biography∼="employment" ∧ Biography∼="landmark"]
8	Person[Name∼="katrina" ∧ Otherworks∼="round"]
9	Person[Otherworks∼="edinburgh" ∧ Otherworks∼="private"]

No.	Query set 6 Query
0	Person[Name∼="ervin"] ⋈ CharacterInMovie
1	Person[Dateofbirth∼="asturias"] ⋈ CharacterInMovie
2	Person[Name∼="ivanova"] ⋈ CharacterInMovie
3	Person[Name∼="ramiro"] ⋈ CharacterInMovie
4	Person[Name∼="maas"] ⋈ CharacterInMovie
5	Person[Name∼="marla"] ⋈ CharacterInMovie
6	Person[Name∼="suzette"] ⋈ CharacterInMovie
7	Person[Name∼="mckinley"] ⋈ CharacterInMovie
8	Person[Name∼="matías"] ⋈ CharacterInMovie
9	Person[Name∼="solange"] ⋈ CharacterInMovie

No.	Query set 7 Query
0	Person[Otherworks∼="forum] ⋈ CharacterInMovie
1	Person[Name∼="moran] ⋈ CharacterInMovie
2	Person[Name∼="kemp] ⋈ CharacterInMovie
3	Person[Name∼="bauer] ⋈ CharacterInMovie
4	Person[Name∼="edna] ⋈ CharacterInMovie
5	Person[Name∼="maxine] ⋈ CharacterInMovie
6	Person[Name∼="kaufman] ⋈ CharacterInMovie
7	Person[Name∼="myriam] ⋈ CharacterInMovie
8	Person[Name∼="hines] ⋈ CharacterInMovie
9	Person[Name∼="artur] ⋈ CharacterInMovie

No.	Query set 8 Query
0	Person[Name∼="barbro"] ⋈ CharacterInMovie ⋈ Movie
1	Person[Name∼="bradshaw"] ⋈ CharacterInMovie ⋈ Movie
2	Person[Name∼="mckee"] ⋈ CharacterInMovie ⋈ Movie
3	Person[Name∼="silas"] ⋈ CharacterInMovie ⋈ Movie
4	Person[Name∼="rachelle"] ⋈ CharacterInMovie ⋈ Movie
5	Person[Otherworks∼="breath"] ⋈ CharacterInMovie ⋈ Movie
6	Person[Name∼="hilliard"] ⋈ CharacterInMovie ⋈ Movie
7	Person[Name∼="redd"] ⋈ CharacterInMovie ⋈ Movie
8	Person[Name∼="jeannine"] ⋈ CharacterInMovie ⋈ Movie
9	Person[Name∼="millard"] ⋈ CharacterInMovie ⋈ Movie

No.	Query set 9 Query
0	Person[Name∼="irena"] ⋈ CharacterInMovie ⋈ Movie
1	Person[Name∼="mohammed] ⋈ CharacterInMovie ⋈ Movie
2	Person[Name∼="davenport"] ⋈ CharacterInMovie ⋈ Movie
3	Person[Name∼="susana"] ⋈ CharacterInMovie ⋈ Movie
4	Person[Biography∼="dangerous"] ⋈ CharacterInMovie ⋈ Movie
5	Person[Name∼="celeste"] ⋈ CharacterInMovie ⋈ Movie
6	Person[Name∼="paco"] ⋈ CharacterInMovie ⋈ Movie
7	Person[Name∼="fabrizio"] ⋈ CharacterInMovie ⋈ Movie
8	Person[Name∼="cliff"] ⋈ CharacterInMovie ⋈ Movie
9	Person[Name∼="orlando"] ⋈ CharacterInMovie ⋈ Movie

A.2.2. Enumeration Test Queries

No.	Query set 2
1	joão,chef/catering
2	prossimo,burgemeester/verteller
3	targout,peiia
4	liberal-left-anarchist,s.burroughs
5	corke,agawa
6	aragon,filmmakers
7	jayenge,burgemeester/verteller
8	captain,borisa
9	five-million,telos
10	gradsko,zaoua

No.	Query set 3
1	exodos,vierikko,couch
2	gradsko,liberal-left-anarchist,domnu
3	verbis,long-forgotten,pig/barnyard
4	saivanidou,moretto,gurralaite
5	testimonios,apogo,matron
6	coppala,deusto,five-million
7	tillbaka,couch,dualitat
8	castalar,sparcy,appartement
9	sanjiro,liberia,giordano
10	jelizawieta,joão,burbank

No.	Query set 4
1	preminchu,disturbassero,exodos,prossimo
2	five-million,cooksen,agawa,seducive
3	aragon,apogo,lepotica,power
4	sparcy,onwura,zlotowas,vezére
5	juanico,misa,effet,liberal-left-anarchist
6	s.burroughs,cooksen,deusto,ancica
7	domnu,grédy,exodos,joão
8	targout,ancica,telos,long-forgotten
9	heat,verbis,long-forgotten,telos
10	s.burroughs,evacuated,castalar,gurralaite

No.	Query set 5
1	castalar,stazione,cnnmoney.com,pushkin,apogo
2	cnnmoney.com,tillbaka,s.burroughs,lepotica,preminchu
3	matron,aragon,modeler,domnu,degan/christian
4	domnu,vierikko,zaoua,telos,joão
5	s.burroughs,liberal-left-anarchist,testimonios,vierikko,tregaye
6	giordano,coppala,seducive,effet,control
7	heat,verbis,long-forgotten,deony,juanico
8	s.burroughs,exodos,souiken,telos,modeler
9	moretto,burbank,control,jelizawieta,twardowska
10	peiia,cutthroat,rivière,sambayanan,apogo

A.2.3. Query execution experiment

Following table shows the queries for the execution experiments. The column "Query" contains the queries. The Column $KWOcc$ illustrates the keyword occurrences, $maxTF$ represents the maximum DF value, $avgDF$ is the average DF value for every keyword in the corresponding query.

Query	KWOcc	maxTF	avgDF
coogee, willfull	3, 3	3, 2	2, 1.3
gobbi, galets	8, 5	22, 4	4.1, 1.6
basha, plaid	10, 17	22, 35	6.2, 8.2
nazarieh, atre, booye	1, 3, 2	1, 3, 1	1, 1.7, 1
iben, blinkende, lygter	8, 5, 5	23, 5, 5	6.9, 2.2, 2.2
rennt, suitcases, tulse	11, 13, 12	7, 30, 41	2.1, 7.6, 10.5
authorize, belosnezhka, nayavu, polyoty	3,4,2,2	7,4,1,1	3.3,3,1,1
gorham, tavis, victrola, norval	12,8,6,9	32,16,3,37	11,8.8,2.2,12.6
aloud, ashanti, longoria, waterford	13,16,12,11	35,19,29,27	8.7,5.3,8.7,12.1

A.2.4. Top-k query sets

No.	Query set 2
0	büttner, unternehmen
1	naroda, georgije
2	vaara, yrjö
3	ejército, morelia
4	insectos, cobo
5	doraemon, daibôken
6	olsenbanden, banden
7	nakedness, advertised
8	rybka, wolski
9	vendicatori, bitto

No.	Query set 3
0	flyday, freleng, friz
1	bocho, monteverde, tamaulipas
2	arrieta, joffe, besito
3	koulussa, finne, jalmari
4	nannies, callison, barcroft
5	conditioned, selzer, bullfighting
6	rikos, kaurismäki, favour
7	kogyaru, hippu, momojiri
8	copia, gervasio, traverso
9	titu, miodrag, kragujevac

No.	Query set 4
0	sizemore, hyams, idolizing, flyboys
1	kamikakushi, hayao, additions, ghibli
2	stuey, advancement, copywriter, strategies
3	hoboes, beaudine, canoga, 215
4	dangan, wakayama, auteurs, dangan
5	perfekt, balk, dials, fairuza
6	família, britto, muniz, brandão
8	streghe, bolognini, pistoia, alphonsine

A.2.5. Concept-based query experiment

Query Set 2
(person:Name:kochan), (person:Realname:kochan)
(person:Name:diler), (position:character:neslisah)
(person:Name:ioannidou), (Drama:title:hristougenna)
(position:character:bõrkabátos), (position:character:görög)
(Drama:title:radovi), (person:Name:zelimir)
(Drama:title:khyana), (person:Name:khalil)
(Short:title:korridorerna), (person:Dateofbirth:västerås)
(Comedy:title:abbuzze), (person:Name:willaert)
(Short:title:owana), (person:Name:horsley)
(Drama:title:fukushû), (Drama:title:chûshingura)
(person:Dateofbirth:ceará), (person:Dateofbirth:iguatu)
(person:Name:pilpani), (position:character:jamlet)
(person:Name:lemken), (Short:title:pokalfieber)
(Adult:title:buraco), (person:Dateofbirth:poá)
(Romance:title:transo), (person:Name:noya)

Query Set 3
(person:Name:relangi), (person:Name:venkatramaiah), (person:Dateofbirth:andhra)
(person:Name:gamlet), (person:Name:khani), (position:character:afandi) (person:Name:irjala), (person:Name:pentti), (Romance:title:järjestää)
(Romance:title:hanawa), (person:Name:junzo), (person:Name:sone)
(Short:title:giorti), (person:Name:spyropoulou), (person:Name:yanna)
(Drama:title:heiya), (person:Name:yasushi), (person:Dateofbirth:asahikawa)
(Comedy:title:ekhay), (person:Name:dyomin), (person:Realname:dyomin)
(Documentary:title:légendes), (person:Name:helmich), (person:Name:naaijkens)
(Drama:title:studer), (Crime:title:kriminalassistent), (Drama:title:wachtmeister)
(person:Name:nováková), (person:Dateofbirth:slavicín), (person:Biography:famu)
(person:Name:lizzi), (person:Otherworks:agoura), (position:character:newsroom)
(person:Name:resino), (person:Dateofbirth:velada), (Short:title:diván)
(Drama:title:absurdistan), (person:Name:allahyari), (person:Name:houchang)
(Short:title:présentation), (person:Name:guillemot), (person:Realname:perche)
(Short:title:sintonía), (person:Name:goenaga), (person:Dateofbirth:donostia)

Query Set 4
(person:Name:ighodaro), (person:Name:osas), (person:Otherworks:collard), (person:Otherworks:platanos)
(person:Name:vrienten), (person:Dateofbirth:hilvarenbeek), (person:Biography:diminishing), (position:character:bioscoop)
(person:Dateofbirth:duluth), (person:Biography:baez), (person:Biography:blowin), (Thriller:title:catchfire)
(Drama:title:variola), (person:Name:goran), (person:Biography:financially), (person:Biography:followup)
(Documentary:title:katei), (person:Name:tichenor), (person:Otherworks:blankets), (person:Otherworks:cinetel)
(Drama:title:rosada), (person:Biography:courtly), (person:Biography:eyesight), (person:Biography:kodama)
(Comedy:title:vami), (person:Name:yekelchik), (person:Realname:izrailevich), (person:Realname:yekelchik)
(Short:title:nukes), (person:Biography:downers), (person:Otherworks:deadender), (person:Otherworks:excessive)
(Drama:title:aigles), (History:title:agonie), (History:title:aigles), (Drama:title:agonie)
(person:Otherworks:blakely), (person:Otherworks:cusack), (person:Otherworks:filumena), (person:Otherworks:gurnett)
(person:Biography:74th), (person:Biography:750), (person:Biography:admired), (position:character:mosby)
(person:Name:shulman), (person:Otherworks:newsletter), (person:Otherworks:publishes), (Short:title:overcoming)
(Short:title:lechu), (person:Name:khrzhanovskiy), (person:Realname:khrzhanovskiy), (person:Realname:yurevich)
(Drama:title:takhti), (person:Name:hassandoost), (person:Biography:hassandoost), (person:Biography:hatami)
(Comedy:title:guardiamarinas), (person:Name:masó), (person:Realname:masó), (person:Realname:paulet)

A.2.6. Effectivness Experiment Results

Person[Trivia ∼= "gyllenhaal", Trivia ∼= "ledger"]
Person[Trivia ∼= "gyllenhaal", Biography ∼= "ledger"]
Person[Biography ∼= "gyllenhaal", Trivia ∼= "ledger"]
Person[Biography ∼= "gyllenhaal", Biography ∼= "ledger"]
Person[Trivia ∼= "gyllenhaal", Name ∼= "ledger"]
Person[Name ∼= "gyllenhaal", Trivia ∼= "ledger"]
Person[Name ∼= "gyllenhaal", Biography ∼= "ledger"]
Person[Realname ∼= "gyllenhaal", Trivia ∼= "ledger"]
Person[Realname ∼= "gyllenhaal", Biography ∼= "ledger"]
Person[Trivia ∼= "gyllenhaal", Realname ∼= "ledger"]
Movie[Title ∼= "ledger"] ⋈ CharacterInMovie ⋈ Person[Trivia ∼= "gyllenhaal"]
MoviePlot[Plot ∼= "ledger"] ⋈ Movie ⋈ CharacterInMovie ⋈ Person[Trivia ∼= "gyllenhaal"]
Person[Trivia ∼= "ledger"] ⋈ CharacterInMovie ⋈ Movie[Title ∼= "gyllenhaal"]
Person[Trivia ∼= "ledger"] ⋈ CharacterInMovie⋈ Movie⋈ WriterInMovie ⋈ Person[Trivia ∼= "gyllenhaal"]
Person[Trivia ∼= "ledger"]⋈ WriterInMovie ⋈ Movie⋈ CharacterInMovie ⋈ Person[Trivia ∼= "gyllenhaal"]

Figure A.2.: Result of query “gyllenhall ledger”

Person[Trivia ∼= "gyllenhaal", Trivia ∼= "ledger"]
Person[Trivia ∼= "gyllenhaal", Biography ∼= "ledger"]
Person[Biography ∼= "gyllenhaal", Trivia ∼= "ledger"]
Person[Biography ∼= "gyllenhaal", Biography ∼= "ledger"]
Person[Trivia ∼= "gyllenhaal", Name ∼= "ledger"]
Person[Name ∼= "gyllenhaal", Trivia ∼= "ledger"]
Person[Name ∼= "gyllenhaal", Biography ∼= "ledger"]
Person[Realname ∼= "gyllenhaal", Trivia ∼= "ledger"]
Person[Trivia ∼= "gyllenhaal", Realname ∼= "ledger"]
Person[Realname ∼= "gyllenhaal", Biography ∼= "ledger"]
Person[Trivia ∼= "ledger"] ⋈ WriterInMovie ⋈ Movie ⋈ CharacterInMovie⋈ Person[Trivia ∼= "gyllenhaal"]
Person[Trivia ∼= "ledger"] ⋈ CharacterInMovie ⋈ Movie ⋈ WriterInMovie ⋈ Person[Trivia ∼= "gyllenhaal"]
Person[Trivia ∼= "ledger"] ⋈ CharacterInMovie ⋈ Movie ⋈ CharacterInMovie ⋈ Person[Trivia ∼= "gyllenhaal"]
Person[Trivia ∼= "ledger"] ⋈ CharacterInMovie ⋈ Movie ⋈ ProducerInMovie ⋈ Person[Trivia ∼= "gyllenhaal"]
Person[Trivia ∼= "ledger"] ⋈ CharacterInMovie ⋈ Movie ⋈ DirectorInMovie ⋈ Person[Trivia ∼= "gyllenhaal"]

Figure A.3.: Result of query “(person::ledger), (person::gyllenhaal)”

Person[Name ∼= "ledger"] ⋈ CharacterInMovie ⋈Movie ⋈ CharacterInMovie ⋈ Person[Name ∼= "gyllenhaal"]
Person[Name ∼= "ledger"] ⋈ CharacterInMovie ⋈Movie ⋈ CharacterInMovie ⋈ Person[Realname ∼= "gyllenhaal"]
Person[Realname ∼= "ledger"] ⋈ CharacterInMovie ⋈ Movie ⋈ CharacterInMovie ⋈ Person[Name ∼= "gyllenhaal"]
Person[Realname ∼= "ledger"] ⋈ CharacterInMovie ⋈ Movie ⋈ CharacterInMovie⋈ Person[Realname ∼= "gyllenhaal"]

Figure A.4.: Result of query “(:name:ledger),(:name:gyllenhaal)”

Person[Name ∼= "ledger"] ⋈ CharacterInMovie ⋈Movie ⋈ CharacterInMovie ⋈ Person[Name ∼= "gyllenhaal"]
Person[Name ∼= "ledger"] ⋈ CharacterInMovie ⋈Movie ⋈ CharacterInMovie ⋈ Person[Realname ∼= "gyllenhaal"]
Person[Realname ∼= "ledger"] ⋈ CharacterInMovie ⋈ Movie ⋈ CharacterInMovie ⋈ Person[Name ∼= "gyllenhaal"]
Person[Realname ∼= "ledger"] ⋈ CharacterInMovie ⋈ Movie ⋈ CharacterInMovie⋈ Person[Realname ∼= "gyllenhaal"]

Figure A.5.: Result of query "(person:name:ledger), (person:name:gyllenhaal)"

Movie[Title ∼= "trek"] ⋈ ProducerInMovie ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ ProducerInMovie ⋈ Person[Realname ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ CharacterInMovie ⋈ Person[Realname ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ MovieReference ⋈ Movie ⋈ CharacterInMovie ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ MovieReference ⋈ Movie ⋈ CharacterInMovie ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ MovieReference ⋈ Movie ⋈ ProducerInMovie ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ MovieReference ⋈ Movie ⋈ ProducerInMovie ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ MovieReference ⋈ Movie ⋈ ProducerInMovie ⋈ Person[Realname ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ MovieReference ⋈ Movie ⋈ ProducerInMovie ⋈ Person[Realname ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ MovieReference ⋈ Movie ⋈ CharacterInMovie ⋈ Person[Realname ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ MovieReference ⋈ Movie ⋈ WriterInMovie ⋈ Person[Realname ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ MovieReference ⋈ Movie ⋈ CharacterInMovie ⋈ Person[Realname ∼= "stewart"]

Figure A.6.: Result of query "(sci-fi:title:trek), (person:name:stewart)"

Movie[Title $\sim=$ "trek"] $\bowtie$ ProducerInMovie $\bowtie$ Person[Name $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ CharacterInMovie $\bowtie$ Person[Name $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ ProducerInMovie $\bowtie$ Person[Realname $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ CharacterInMovie $\bowtie$ Person[Realname $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ MovieReference $\bowtie$ Movie $\bowtie$ CharacterInMovie $\bowtie$ Person[Name $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ MovieReference $\bowtie$ Movie $\bowtie$ CharacterInMovie $\bowtie$ Person[Name $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ MovieReference $\bowtie$ Movie $\bowtie$ ProducerInMovie $\bowtie$ Person[Name $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ MovieReference $\bowtie$ Movie $\bowtie$ ProducerInMovie $\bowtie$ Person[Name $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ MovieReference $\bowtie$ Movie $\bowtie$ ProducerInMovie $\bowtie$ Person[Realname $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ MovieReference $\bowtie$ Movie $\bowtie$ ProducerInMovie $\bowtie$ Person[Realname $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ MovieReference $\bowtie$ Movie $\bowtie$ CharacterInMovie $\bowtie$ Person[Realname $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ MovieReference $\bowtie$ Movie $\bowtie$ WriterInMovie $\bowtie$ Person[Realname $\sim=$ "stewart"]
Movie[Title $\sim=$ "trek"] $\bowtie$ MovieReference $\bowtie$ Movie $\bowtie$ CharacterInMovie $\bowtie$ Person[Realname $\sim=$ "stewart"]

Figure A.7.: Result of query "(drama:title:trek), (person:name:stewart)"

Person[Name ~= "stewart", Trivia ~= "trek"]
Person[Name ~= "stewart", Biography ~= "trek"]
Person[Otherworks ~= "stewart", Trivia ~= "trek"]
Person[Otherworks ~= "stewart", Biography ~= "trek"]
MoviePlot[Plot ~= "stewart", Plot ~= "trek"]
Person[Name ~= "stewart", Otherworks ~= "trek"]
Person[Trivia ~= "stewart", Trivia ~= "trek"]
Person[Biography ~= "stewart", Trivia ~= "trek"]
Person[Trivia ~= "stewart", Biography ~= "trek"]
Person[Biography ~= "stewart", Biography ~= "trek"]
Person[Otherworks ~= "stewart", Otherworks ~= "trek"]
Person[Realname ~= "stewart", Trivia ~= "trek"]
Person[Realname ~= "stewart", Biography ~= "trek"]
Person[Trivia ~= "stewart", Otherworks ~= "trek"]
Person[Biography ~= "stewart", Otherworks ~= "trek"]

Figure A.8.: Result of query “trek, stewart”

Movie[Title ∼= "star", Title ∼= "trek"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "star", Title ∼= "trek"] ⋈ ProducerInMovie ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "star", Title ∼= "trek"] ⋈ ProducerInMovie ⋈ Person[Realname ∼= "stewart"]
Movie[Title ∼= "star", Title ∼= "trek"] ⋈ CharacterInMovie ⋈ Person[Realname ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ ProducerInMovie ⋈ MovieReference ⋈ Movie[Title ∼= "star"] ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "stewart"] ⋈ MovieReference ⋈ Movie[Title ∼= "star"]
Movie[Title ∼= "trek"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "stewart"] ⋈ MovieReference ⋈ Movie[Title ∼= "star"]
Movie[Title ∼= "trek"] ⋈ CharacterInMovie ⋈ MovieReference ⋈ Movie[Title ∼= "star"] ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ ProducerInMovie ⋈ MovieReference ⋈ Person[Name ∼= "stewart"] ⋈ Movie[Title ∼= "star"]
Movie[Title ∼= "trek"] ⋈ ProducerInMovie ⋈ MovieReference ⋈ Person[Name ∼= "stewart"] ⋈ Movie[Title ∼= "star"]
Movie[Title ∼= "trek"] ⋈ CharacterInMovie ⋈ MovieReference ⋈ Movie[Title ∼= "star"] ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ ProducerInMovie ⋈ MovieReference ⋈ Movie[Title ∼= "star"] ⋈ Person[Name ∼= "stewart"]
Movie[Title ∼= "trek"] ⋈ CharacterInMovie ⋈ MovieReference ⋈ Person[Name ∼= "stewart"] ⋈ Movie[Title ∼= "star"]
Movie[Title ∼= "trek"] ⋈ CharacterInMovie ⋈ MovieReference ⋈ Person[Name ∼= "stewart"] ⋈ Movie[Title ∼= "star"]
Movie[Title ∼= "trek"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "stewart"] ⋈ CharacterInMovie ⋈ Movie[Title ∼= "star"]

Figure A.9.: Result of query “(sci-fi:title:star), (sci-fi:title:trek), (person:name: stewart)”

Person[Name ∼= "kubrick"] ⋈ DirectorInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ WriterInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ ProducerInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ ProducerInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Realname ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ DirectorInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Realname ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ WriterInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Realname ∼= "sellers"]

Figure A.10.: Result of query “(person:name:kubrick), (movie:title:strangelove), (person:name:sellers)”

CharacterInMovie[CharacterName ∼= "kubrick"] ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie[CharacterName ∼= "sellers"]
CharacterInMovie[CharacterName ∼= "kubrick"] ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ ProducerInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ WriterInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ DirectorInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Name ∼= "sellers"]
CharacterInMovi[CharacterName ∼= "kubrick"] ⋈ Person ⋈ CharacterInMovie Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie [CharacterName ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ DirectorInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Realname ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ ProducerInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Realname ∼= "sellers"]
Person[Name ∼= "kubrick"] ⋈ WriterInMovie ⋈ Movie[Title ∼= "strangelove"] ⋈ CharacterInMovie ⋈ Person[Realname ∼= "sellers"]

Figure A.11.: Result of query “(:name:kubrick), (:title:strangelove), (:name:sellers)”

Person[Biography ∼= "sellers", Trivia ∼= "strangelove", Trivia ∼= "kubrick"]
Person[Trivia ∼= "sellers", Trivia ∼= "strangelove", Trivia ∼= "kubrick"]
Person[Biography ∼= "sellers", Trivia ∼= "strangelove", Biography ∼= "kubrick"]
Person[Trivia ∼= "sellers", Trivia ∼= "strangelove", Biography ∼= "kubrick"]
Person[Name ∼= "sellers", Trivia ∼= "strangelove", Biography ∼= "kubrick"]
Person[Trivia ∼= "sellers", Biography ∼= "strangelove", Trivia ∼= "kubrick"]
Person[Biography ∼= "sellers", Biography ∼= "strangelove", Biography ∼= "kubrick"]
Person[Trivia ∼= "sellers", Biography ∼= "strangelove", Biography ∼= "kubrick"]
Person[Name ∼= "sellers", Biography ∼= "strangelove", Biography ∼= "kubrick"]
Person[Otherworks ∼= "sellers", Trivia ∼= "strangelove", Biography ∼= "kubrick"]
MoviePlot[Plot ∼= "sellers", Plot ∼= "strangelove", Plot ∼= "kubrick"]
Person[Otherworks ∼= "sellers", Biography ∼= "strangelove", Biography ∼= "kubrick"]
Person[Realname ∼= "sellers", Trivia ∼= "strangelove", Biography ∼= "kubrick"]
Person[Trivia ∼= "sellers", Trivia ∼= "strangelove", Name ∼= "kubrick"]
Person[Realname ∼= "sellers", Biography ∼= "strangelove", Biography ∼= "kubrick"]

Figure A.12.: Result of query “kubrick, strangelove, sellers”