Olaf Görlitz

Distributed Query Processing for Federated RDF Data Management

vom Promotionsausschuss des Fachbereichs 4: Informatik der Universität Koblenz–Landau zur Verleihung des akademischen Grades Doktor der Naturwissenschaften (Dr. rer. nat.) genehmigte Dissertation.

Datum der wissenschaftlichen Aussprache: 07.11.2014

Vorsitzender des Promotionsausschusses: Prof. Dr. Ralf Lämmel

Promotionskommission

Vorsitzende:	Prof. Dr. Viorica Sofronie-Stokkermans
Berichterstatter:	Prof. Dr. Steffen Staab
	Prof. Dr. Jürgen Ebert
	Prof. Dr. Georg Lausen

Abstract

The publication of freely available and machine-readable information has increased significantly in the last years. Especially the Linked Data initiative has been receiving a lot of attention. Linked Data is based on the Resource Description Framework (RDF) and anybody can simply publish their data in RDF and link it to other datasets. The structure is similar to the World Wide Web where individual HTML documents are connected with links. Linked Data entities are identified by URIs which are dereferenceable to retrieve information describing the entity. Additionally, so called SPARQL endpoints can be used to access the data with an algebraic query language (SPARQL) similar to SQL. By integrating multiple SPARQL endpoints it is possible to create a federation of distributed RDF data sources which acts like one big data store.

In contrast to the federation of classical relational database systems there are some differences for federated RDF data. RDF stores are accessed either via SPARQL endpoints or by resolving URIs. There is no coordination between RDF data sources and machine-readable meta data about a source's data is commonly limited or not available at all. Moreover, there is no common directory which can be used to discover RDF data sources or ask for sources which offer specific data.

The federation of distributed and linked RDF data sources has to deal with various challenges. In order to distribute queries automatically, suitable data sources have to be selected based on query details and information that is available about the data sources. Furthermore, the minimization of query execution time requires optimization techniques that take into account the execution cost for query operators and the network communication overhead for contacting individual data sources. In this thesis, solutions for these problems are discussed. Moreover, SPLENDID is presented, a new federation infrastructure for distributed RDF data sources which uses optimization techniques based on statistical information.

Zusammenfassung

Die weltweite Vernetzung von semantischen Information schreitet stetig voran und erfährt mit der Linked Data Initiative immer mehr Aufmerksamkeit. Bei Linked Data werden verschiedene Datensätze aus unterschiedlichen Domänen und von diversen Anbietern in einem einheitlichen Format (RDF) zur Verfügung gestellt und miteinander verknüpft. Strukturell ist das schnell wachsende Linked Data Netzwerk sehr ähnlich zum klassischen World Wide Web mit seinen verlinkten HTML Seiten. Bei Linked Data handelt es sich jedoch um URI-referenzierte Entitäten, deren Eigenschaften und Links durch RDF-Triple ausgedrückt werden. Neben dem Dereferenzieren von URIs besteht mit SPARQL auch die Möglichkeit, ähnlich wie bei Datenbanken, komplexe algebraische Anfragen zu formulieren und über sogenannte SPARQL Endpoints auf einer Datenquelle auswerten zu lassen. Eine SPARQL Anfrage über mehrere Linked Data Quellen ist jedoch kompliziert und bedarf einer föderierten Infrastruktur in der mehrere verteilte Datenquellen integriert werden, so dass es nach außen wie eine einzige große Datenquelle erscheint.

Die Föderation von Linked Data hat viele Ähnlichkeiten mit verteilten und föderierten Datenbanken. Es gibt aber wichtige Unterschiede, die eine direkte Adpation von bestehenden Datenbanktechnologien schwierig machen. Dazu gehört unter anderem die große Anzahl heterogener Datenquellen in der Linked Data Cloud, Beschränkungen von SPARQL Endpoints, und die teils starke Korrelation in den RDF Daten. Daher befasst sich die vorliegende Arbeit primär mit der Optimierung von verteilten SPARQL Anfragen auf föderierten RDF Datenquellen. Die Grundlage dafür ist SPLENDID, ein effizientes Optimierungverfahren für die Ausführung von verteilten SPARQL Anfragen in einer skalierbaren und flexiblen Linked Data Föderationsinfrastruktur. Zwei Aspekte sind dabei besonders wichtig: die automatische Auswahl von passenden Datenquellen für beliebige SPARQL Anfragen und die Berechnung des optimalen Ausführungsplans (Join Reihenfolge) basierend auf einem Kostenmodell. Die dafür erforderlichen statistischen Information werden mit Hilfe von VOID-basierten Datenquellenbeschreibungen zur Verfügung gestellt. Darüberhinaus wird auch des Management verteilter statistischer Daten untersucht und eine Benchmark-Methodologie für föderierte SPARQL Anfragen präsentiert.

Acknowledgements

I want to thank Steffen Staab for his extended support and guidance in my research. He was always helpful, gave valuable advice when I seemed to get stuck, and could always show me ways to further improve the research results. I would also like to thank my colleagues from the Institute of Web Science and Technologies at the University of Koblenz-Landau for many fruitful discussions, stimulating ideas, and their constructive feedback. Especially, the support and persistent encouragement of Matthias Thimm helped me a lot to take pride in my research and to complete important parts of my work. In addition, I want to thank Simon Schenk, Carsten Saathoff, and Thomas Franz for giving valuable feedback on initial stages of my research. Moreover, I always enjoyed the familiar atmosphere in the research group and many colleagues became friends.

Furthermore, I would like to thank the fellow researchers in the project CollabCloud for interesting discussions, useful feedback for improving SPLENDID, for exchanging ideas which led to the development of SPLODGE, and the appreciation of my work. I'm also thankful for the cooperation with the colleagues in the EU project Tagora. They introduced me to the interesting research area of complex systems and made my research in the direction of distributed tagging systems quite enjoyable.

Finally, I want to thank my parents for their encouragement and support in all my endeavors. Special thanks to my brother Roland who gave me valuable feedback from an outside perspective. I'm also very grateful for Kerstin's support, especially at times when writing the dissertation did not proceed as expected.

Contents

1 Introduction 1
1.1 Scenario 3
1.2 Research Challenges 4
1.3 Research Contributions 6
1.4 Overview 7

2 Structured Data on the Web 9
2.1 RDF 9
2.1.1 IRIs, URIrefs, Literals, and Blank Nodes 10
2.1.2 RDF Graphs 10
2.1.3 RDF Vocabularies 11
2.1.4 RDF Syntax 12
2.1.5 Named Graphs 13
2.2 Linked Open Data 14
2.2.1 Linked Data Principles 14
2.2.2 Linked Data Cloud 16
2.2.3 Consuming Linked Data 17
2.2.4 Limitations of Linked Open Data 17
2.3 SPARQL 17
2.3.1 SPARQL 1.1 20
2.3.2 SPARQL Federation 20
2.4 RDF Stores 21
2.4.1 RDF in RDBMS 21
2.4.2 Column Stores 22
2.4.3 Triple and Quad Stores 23
2.4.4 Indexing RDF 24
2.4.5 Relational Data Integration 25
2.5 Graph Technologies 26
2.5.1 Graph Algorithms for RDF 26
2.5.2 Graph Model Transformation 26
2.5.3 Graph Databases 27
2.6 Summary 28

3 Distributed RDF Data Source Integration . . . 29
3.1 Linked Data Integration Scenarios . . . 30
3.1.1 System Requirements . . . 30
3.1.2 Classification of Infrastructure Paradigms . . . 33
3.2 Data Warehousing . . . 35
3.2.1 Materializing Linked Data . . . 35
3.2.2 Data Clustering . . . 35
3.3 Search Engines . . . 36
3.3.1 Crawling and Indexing . . . 36
3.3.2 Ranked Retrieval . . . 37
3.4 Federated Systems . . . 38
3.4.1 Mediator-based Architecture . . . 38
3.4.2 Data Source Management . . . 39
3.4.3 Distributed Query Processing . . . 41
3.5 Link Traversal Query Processing . . . 44
3.5.1 Non-blocking Query Execution . . . 44
3.5.2 Limitations of Pure Linked Data Queries . . . 45
3.6 Peer-to-Peer Systems . . . 47
3.6.1 Overlay Network Topologies . . . 47
3.6.2 RDF in Structured Peer-to-Peer Networks . . . 48
3.7 Event-based Middleware . . . 50
3.7.1 RDF Stream Processing . . . 50
3.7.2 Publish/Subscribe Systems . . . 51
3.8 Summary . . . 51

4 SPLENDID Data Source Selection . . . 53
4.1 Federated Linked Data Management . . . 54
4.1.1 Static and Dynamic Data Source Integration . . . 54
4.1.2 Describing Data Sources with VOID . . . 55
4.1.3 VOID Generation . . . 56
4.2 Data Source Indexing . . . 57
4.2.1 Schema-level Index . . . 58
4.2.2 Instance-level Index . . . 59
4.2.3 Graph-level Index . . . 60
4.3 Data Source Selection . . . 61
4.3.1 Schema-based Source Selection . . . 62
4.3.2 Data Source Pruning with ASK Queries . . . 62
4.3.3 Sub-Query Optimization . . . 64
4.4 Evaluation . . . 65
4.4.1 Setup . . . 66
4.4.2 Results . . . 67
4.5 Summary . . . 68

5 SPLENDID Cost-based Distributed Query Optimization 71
5.1 Query Optimization in Traditional Databases 72
5.1.1 Query Plans 72
5.1.2 Join Algorithms 74
5.1.3 Query Optimization Strategies 76
5.2 Query Optimization Challenges for Federated Linked Data ... 78
5.2.1 Optimization Objectives 78
5.2.2 Comparison of Linked Data with Federated Databases . 80
5.2.3 Distributed SPARQL Query Optimization Strategies ... 81
5.2.4 Maintaining Linked Data Statistics 83
5.3 Join Order Optimization for Distributed SPARQL Queries 84
5.3.1 Linked Data Integration Requirements 85
5.3.2 SPARQL Join Implementations 85
5.3.3 Cost-based Join Order Optimization 88
5.3.4 Representing RDF Statistics with VOID 90
5.3.5 A Cost Model for Distributed SPARQL Queries 92
5.4 Cardinality Estimation for Distributed SPARQL Queries 94
5.4.1 Triple Pattern Cardinality Estimation 94
5.4.2 Cardinality of Basic Graph Patterns 97
5.5 Evaluation 100
5.5.1 Setup 102
5.5.2 Results 103
5.6 Summary 105

6 PINTS: Maintaining Distributed Statistics for Peer-to-Peer Information Retrieval 107
6.1 A Data Model for Collaborative Tagging Systems 108
6.1.1 Tripartite Networks 109
6.1.2 Generic Tag Clouds 110
6.1.3 Feature Vectors for Folksonomies 111
6.2 Managing Distributed Tagging Data 114
6.2.1 Decentralized Storage of Tagging Data 116
6.2.2 Feature Vector based Information Retrieval 116
6.3 Efficient Updates for Distributed Statistics 118
6.3.1 Feature Vector Update Strategies 118
6.3.2 Prediction-based Feature Vector Approximation 120
6.4 Evaluation 123
6.4.1 Setup 123
6.4.2 Results 124
6.5 Summary 127

7 SPLODGE Benchmark Methodology for Federated Linked Data Systems ... 129
7.1 Design Issues for Federated RDF Benchmarks ... 130
7.1.1 Datasets ... 130
7.1.2 Queries ... 132
7.1.3 Evaluation Environment ... 133
7.2 SPARQL Query Characterization ... 134
7.2.1 Algebra ... 136
7.2.2 Structure ... 136
7.2.3 Cardinality ... 137
7.3 Query Generation Methodology ... 138
7.3.1 Query Parameterization ... 138
7.3.2 Iterative Query Generation ... 139
7.3.3 Validation of Generated Queries ... 144
7.4 Evaluation ... 144
7.4.1 Setup ... 145
7.4.2 Results ... 151
7.5 Summary ... 155

8 Conclusion ... 157
8.1 Summary and Research Contributions ... 158
8.2 Research Obstacles and Lessons Learned ... 159
8.3 Outlook and Future Work ... 161

References ... 163

Index ... 179

List of Figures

2.1 RDF example describing a publication by Paul Erdős 11
2.2 Graph representation of the running example 11
2.3 RDF/XML representation of the running example 12
2.4 N-Triples representation of the running example 13
2.5 RDFa example with embedded RDF data 14
2.6 Classification of Private, Public and Open Data 15
2.7 The Linked Open Data Cloud 16
2.8 SPARQL query forms 18
2.9 SPARQL result sets 19
2.10 SPARQL query example: German co-authors of Paul Erdős 19
2.11 Information about Paul Erdös in three linked data sources 21
2.12 Federated SPARQL query with keywords SERVICE and VALUES . . 21
2.13 RDF data stored in different table layouts 23

3.1 Classification of data integration paradigms 33
3.2 Overview of the SPLENDID federation architecture 39

4.1 Example VOID description for the ChEBI dataset 56
4.2 A SPARQL query with derived data source mappings 61
4.3 Comparison of selected data sources and number of requests . . . 67

5.1 A SPARQL query represented as left-deep tree and bushy tree . . 73
5.2 Trade-off visualization for accuracy vs. maintenance cost 84
5.3 Variations of Semi-Join variable bindings in SPARQL queries . . . 87
5.4 VOID statistics for the Drugbank dataset 91
5.5 Performance comparison for separate Hash-Join and Bind-Join . 104
5.6 Performance comparison for combined Hash-Join and Bind-Join 105
5.7 Comparison of SPLENDID with DARQ, FedX, and AliBaba 106

6.1 Visualization of tag assignment relations 109
6.2 Tag cloud example 110
6.3 Tag-centered visualization of five tag assignments by three users 114
6.4 Screenshot of Tagster 115

6.5 Comparison of feature vector similarity violations 125
6.6 Comparison of the number of update messages 126

7.1 Overview of graph patterns in FedBench queries 135
7.2 Query Structure Generation Example 139
7.3 Pre-processing steps for generating the datasets statistics 147
7.4 Comparison of SPLODGE*lite* and SPLODGE 152
7.5 Comparison of estimated and real result cardinality 153

List of Tables

3.1 Comparison of infrastructure paradigms 34
3.2 Comparison of Semantic Search Engines 37
3.3 Overview of federation systems 43
3.4 Overview of link traversal approaches 46

4.1 FedBench evaluation dataset statistics 66
4.2 Data source coverage and result size of FedBench queries 66

5.1 Optimizer characteristics of state-of-the-art federation systems . 101
5.2 Details of evaluated FedBench queries 102

6.1 Statistics of the Flickr and Delicious tagging datasets 123

7.1 RDF benchmark query features 133
7.2 Statistics of different Billion Triple Challenge Datasets 146
7.3 Number of Entities and Blank Nodes in the 2011 BTC Dataset . 148
7.4 Resource Predicate Index 149
7.5 Predicate Path Statistics 149
7.6 Characteristic Set Statistics 150
7.7 Ranking of predicates in generated benchmark queries 154

1

Introduction

The amount of information published on the *World Wide Web* is constantly increasing. Today's web search engines employ sophisticated full-text indexing and keyword-based search in oder to quickly find relevant documents from the billions of web sites. But neither are they able to cover information from the *Deep Web*, i. e. data stored in databases and hidden behind dynamic web front-ends, nor can they connect information from different web sources. In order to master current and future information it is necessary to employ automatic processing and interpretation of machine-readable data. *Microformats*[1], *Schema.org*[2], and *MicroData*[3] are recent examples for expressing structured content, i. e. resources with specific properties, inside web pages. They are recognized by the major search engines and allow for better interpretation of presentation of web resources with specific properties, e. g. consumer products. However, these approaches only provide a specific markup syntax and fixed interpretation rules. In fact, large amounts of data from the *Deep Web* can hardly be exposed through individual microformats. Moreover, the data aggregation from different sources is not sufficiently supported.

The objective of the *Linking Open Data* initiative[4] is to open up data silos and publish structured data in a machine-readable format with typed links between related resources. As a result a growing number of various datasets from different domains are made available which can be freely browsed and searched to find and extract useful information. The network of interlinked datasets, i. e. the so called *Linked Data Cloud*, has a structure similar to the web itself where web sites are connected by hyperlinks. Each Linked Data

[1] http://microformats.org/
[2] http://schema.org/
[3] http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html
[4] http://esw.w3.org/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

resource is uniquely identified by a URI. The description of resources and the relationships between resources are expressed with the *Resource Description Framework* (RDF) [202].

Basically everybody can publish datasets as Linked (Open) Data. The *linked data principles* [27] only require that resource URIs are resolvable, i. e. a common HTTP GET request on the URI returns *useful information* about the resource. In order to improve the quality and benefits of datasets, the Linked Data guidelines include specific recommendations, like the reuse of common vocabularies. Moreover, the Linked Data Patterns [69] encouraged a consistent use of modeling schemes. A five star Linked Data classification schema[5] gives hints on how to stepwise improve the quality and openness of Linked Data. A good overview about the foundations and related topics of Linked Open Data is given in [30].

Finding relevant information in the Linked Data cloud is challenging, not only because of its size but also because of its open nature and the varying quality of the datasets. There are two orthogonal approaches for consuming Linked Open Data: (1) keyword-based search for entities, similar to traditional web search and (2) the evaluation of complex algebraic queries, like in databases, using the whole Linked Data cloud as a huge information space. The first case typically occurs when a user is not searching for a specific piece of information but first wants to learn about different resources in an area of interest. Keyword-based search and link exploration helps to narrow down the search space. This essentially leads to an iterative search process and new information can be discovered while following links and expanding the search scope. In contrast, when a user has a good understanding of the actual data schema and vocabularies used, it is possible to formulate complex queries which describe specific resource properties and relationships between resources that should be returned as a result.

In order to leverage the full potential of the Linked Data cloud a user needs to access all the data in a flexible way. Therefore, simple querying of various data sources and automatic processing of distributed data should be offered while hiding the complexity of the data aggregation, i. e. the integration of various data sources and the mapping of vocabulary and data schemas is fully transparent to the user. However, a common query interface for Linked Data which fulfills all these requirements is currently not available. Instead, single data sources can be queried through so called *SPARQL endpoints*. These query interfaces evaluate SPARQL queries on the local RDF graph and return the results to the caller. The integration of many Linked Data sources with SPARQL endpoints behind a common query interface can be realized through a federation infrastructure. Linked Data federation has certain advantages over other data integration approaches (cf. Chap. 3). Therefore, this dissertation focuses on the challenges for Linked Data federation and distributed query processing.

[5] http://www.w3.org/DesignIssues/LinkedData.html

1.1 Scenario

Getting a general overview of a specific research area is often a tedious task. It requires, for example, exploring current state-of-the-art, spotting trending topics, identifying the most important research questions, finding highly cited scientific publications, and ranking the main conferences in the area. Moreover, it may be interesting to get an overview about public funding for research projects and to identify social connections between researchers, e. g. as derived from joint publications. Finally, filtering and ranking, e. g. based on number of publications with high impact or visibility, is necessary to provide a 'big picture' of the hot topics and prominent researchers.

Consider a PhD student who is looking for a research topic. It typically takes a couple of months to get base knowledge about a research area and to come up with potentially interesting research questions that have not been solved before. A tool which can aggregate all the information mentioned above would be extremely useful. It can provide the necessary overview, including past work in the area, and it would allow for checking whether certain research questions have already been fully covered. Furthermore, the tool can be used to observe current activities and new publications.

In fact, all the information is already available on the web, even as Linked Open Data. There are many different data sources which can be searched for relevant information, e. g. over 1.3 million publications with information about authors and conferences can be found in the *Digital Bibliography and Library Project*[6] (DBLP), the *Community Research and Development Service* (CORDIS) provides data about EU-funded research projects and the participating institutions, DBpedia [15] offers semistructured data for Wikipedia articles, and profiles of researchers are described with the *friend-of-a-friend* (FOAF) vocabulary [37]. Tools like the *CS AKTive Space*[7] project already integrate some of this information but are usually focused on a specific subset, i. e. in this case the UK Computer Science research domain. Instead, we need an infrastructure and a toolset which can provide access to the whole Linked Open Data cloud. Thus, a query like 'give me all important publications for distributed query processing on RDF data, ranked by impact' should return the top papers along with information about authors, institutions and associated research projects, all without the need to define any data sources for the search. The tool automatically gathers and merges all relevant information from the various known data sources and presents a concise overview. While the results are valuable for the PhD student it might also be useful for identifying research institutes when looking for a new job or just to rank the impact or activity of different researchers.

[6] http://dblp.uni-trier.de/
[7] http://www.aktors.org/technologies/csaktivespace/

1.2 Research Challenges

It has been mentioned before, that datasets in the Linked Data cloud are very heterogeneous and of varying quality. There is neither a quality assurance nor any kind of approval process for deciding if a dataset is acceptable. Linked Open Data represents basically a grassroot movement where everybody can just publish their data using RDF and link it to other existing datasets. Hence, there are various problems when dealing with diverse schemas, sparse meta-data, and limitations of query interfaces. In this thesis, the focus lies primarily on the efficient implementation of distributed query processing in a federation of SPARQL endpoints. Thus, the challenges are similar to those for traditional query optimization in (federated) database systems and information retrieval in distributed systems.

A federation system automatically distributes SPARQL queries to remote SPARQL endpoints and aggregates the returned results. The query processing time is significantly influenced by the number of contacted data sources and the amount of data transferred over the network. Hence, in order to avoid contacting all data sources, the federation systems needs information about the data sources and the data they provide. Statistical data is commonly used in databases and can also be applied for Linked Data federation. However, such detailed knowledge is hardly available for the whole Linked Data cloud. Thus, we have to deal with new challenges which limit the adaptation of common database strategies. In general, we differentiate between three important aspects which need to be covered by a federation infrastructure for SPARQL endpoints, namely (1) managing Linked Data sources with (limited) statistical metadata, (2) efficient distributed query processing, and (3) a scalable implementation of the federation infrastructure.

Data Source Management

The Linked Data cloud is constantly growing and datasets change over time. However, a complete list of all available Linked Data sources does not exist and can hardly be compiled. Nevertheless, the *CKAN data hub*[8] collects information information about freely available Linked Data sources. But it is far from complete and only basic meta-data is provided for the listed data sources. Hence the federation infrastructure must implement specific mechanisms to obtain the information required for data sources selection and query optimization. The Vocabulary for Interlinked Datasets (VoiD) [9] allows for describing data sources. But it is not widely used yet and data providers may support it at different levels of granularity. Hence, it is necessary to have a mechanism which can obtain the required statistical information about data sources. Typically, the statistical information should have a high level of detail in order to obtain accurate estimation results during query optimization. However, detailed statistics also imply more space for data storage which

[8] http://thedatahub.org

leads to a trade-off between space consumption and statistical details. Moreover, changes in datasets require that these statistics have to be updated. Otherwise, the query optimization can produce sub-optimal query plans due to wrong cardinality estimations. Since an explicit change notification mechanism does not exist, a federation system has to take care that the statistics are always up-to-date.

Distributed Query Processing

The query processing in a federation environment has to deal with different optimization objectives. Query execution time is typically the most important one but also the amount of transferred data and result completeness play an important role. Especially the network communication overhead has a significant influence on the query execution time. Since only few data sources will typically be able to return results for a query it is important to limit the number of contacted SPARQL endpoints to those which can actually return useful results. In general, the query processing has to deal with two major challenges: (1) identifying suitable data sources which can return results for a given query and (2) applying a query optimization strategy which finds the optimal query execution plan according to query execution time, network communication cost, and result completeness.

Query optimization approaches based on heuristics can hardly handle complex query structures and take all constraints into account. Therefore, a query optimization based on statistical information can often produce better results if the cardinality of the query results can be estimated accurately. Furthermore, due to the heterogeneity of the Linked Data cloud we also have to deal with technical constraints, e. g. limited bandwidth, high latency, and timeouts are typical problems that are encountered. Coping with all these issues requires more advanced strategies, like adaptive query processing. However, this also comes with higher complexity for the query planning. Hence, a static query optimization approach is typically applied.

Implementing and Testing Scalable Linked Data Federation

Currently, there exists no federation system which integrates the whole Linked Data cloud. Moreover, there are also no benchmarks which allow for the objective evaluation of different federation implementations. In fact, there is only limited experience available with typical Linked Data usage scenarios and the accompanied technical challenges. In most cases only a few datasets from certain domains are integrated for specific application scenarios. But such limited use cases cannot be used for testing scalability of a full fledged Linked Data federation infrastructure. In order to evaluate scalability and performance of federation implementations it is necessary to have a realistic setup and typical query scenarios. While typical benchmarks focus mainly on query execution time, this may not be the only aspect which is important for a federation

implementation. The flexibility concerning dataset integration and the adaptation to changing datasets may also play an important role. However, since benchmarks need to be reproducible it is necessary to acquire representative snapshots of the Linked Data cloud which implies other challenges with respect to data size or use of sampling techniques. Furthermore, queries in a federation system have other characteristics than queries in non-federated scenarios. Hence, setting up a federation for Linked Data always requires human interaction, e. g. to select relevant data sources or define interesting query characteristics.

1.3 Research Contributions

The research presented in this thesis deals with the aforementioned challenges in different ways. Following is an overview of the main contributions.

Managing Distributed Data Sources

The integration of different SPARQL endpoints in a federation requires automatic data source discovery, maintenance of relevant meta-data, and data sources management for providing transparent access to the Linked Data cloud. The SPLENDID federation approach relies on statistic information collected from the Linked Data sources in order to decide where relevant data can be found for a given query. VOID descriptions [9] are used for representing the required meta-data and simple statistical information of data sources with a common vocabulary. Heuristics are employed in order to compute an optimal mapping between query expressions and data sources. The results of this source selection approach is important for the effectiveness of the distributed query optimization.

Distributed Query Optimization

The SPLENDID query optimization approach is based on statistical information which are used to estimate cardinality and selectivity of query results. The join-order optimization algorithm employs dynamic programming [205]. In contrast to a purely heuristics-based approach this allows for a better estimation of the individual query operator cost since the estimated size of intermediate results can be taken into account. While this kind of query optimization is already known from traditional databases, SPLENDID takes into account the special characteristics of RDF data which leads to better estimation results, depending on the details of the underlying statistics.

Distributed Statistics for Full-text Search

RDF data exhibits a skewed data distribution where popular predicates and types can be found in almost every data source. PINTS allows for weighting statistical information from different Linked Data sources in order to accommodate for such data distributions. The underlying infrastructure is based on a peer-to-peer network. It distributes and updates aggregated information from all data sources in an efficient way. Based on these shared statistics full-text search can basically be realized for the Linked Data cloud.

Linked Data Benchmarking

Benchmarking is important for testing and comparing different federation implementations. Our contribution is a methodology and a toolset for the systematic generation of SPARQL queries which cover a wide range of possible requests on the Linked Data cloud. A classification of query characteristics provides the basis for the query generation strategy. The query generation heuristic of SPLODGE (SPARQL Linked Open Data Query Generator) employs stepwise combination of query patterns which are selected based on predefined query characteristics, e. g. query structure, result size, and affected data sources.

1.4 Overview

Chapter 2 gives background information about the fundamental technologies which are relevant for the presented work. Then the different architectures and query processing approaches for consuming Linked Data are explained in Chapter 3. Thereafter, SPLENDID is presented in detail, with focus on the data source management and sources selection in Chapter 4 and presenting the statistics-based distributed query optimization in Chapter 5. Chapter 6 continues with the PINTS approach for managing distributed statistic for full-text search. Chapter 7 presents benchmarking Linked Data query processing using the SPLODGE methodology and toolkit. Finally, Chapter 8 concludes and gives and overview of future research directions which have been discovered in this work.

2

Structured Data on the Web

To search the vast amount of information on the World Wide Web, search engines have become indispensable. Keyword-based search allows us to find all kinds of information quickly. However, the content of search results cannot be understood by a search engine but has to be interpreted by a human user. One approach for improving machine readability is to use structured data with semantics and interlinks between related data items. In recent years we can observe the evolution of the World Wide Web into a Web of Data. This also offers more possibilities for information retrieval well beyond keyword search. In the following, we will give an overview on the most important technologies on which the Web of Data is based on.

2.1 RDF

The *Resource Description Framework* (RDF) is a generic, graph-based data model used for describing resources in a machine-readable way. RDF was specified by the *World Wide Web Consortium* (W3C) and it is a widely accepted standard in the Semantic Web. Due to its flexibility it has become a popular format for representing and processing semi-structured data on the Web.

Definition 2.1 (RDF Triple). *Let I, L, and B be the pairwise disjoint infinite sets of IRIs [71], Literals, and Blank nodes, respectively. A triple $(s, p, o) \in \{I \cup B\} \times I \times \{I \cup B \cup L\}$ is called an RDF triple. By convention, s is the subject, p is the predicate, and o is the object of the RDF triple.*

2.1.1 IRIs, URIrefs, Literals, and Blank Nodes

There are two different specifications for identifying resources in RDF. The first W3C Recommendation of RDF [154] introduced RDF URI references[1] because the specification of the *Internationalized Resource Identifiers* [71] (IRIs) was not yet finished. RDF URI references are compatible with IRIs, but there are significant differences concerning the handling of Unicode and special characters, e. g. the characters `<`, `>`, `{`, `}`, `|`, “, `^`, ‘, `"` are allowed in RDF URI references whereas all special character in IRIs must be percent encoded. These differences can result in compatibility problems, especially since the SPARQL query language specification (cf. Sec. 2.3) is based on IRIs. Hence, the RDF 1.1 specification [60], which supersedes the previous version, replaces RDF URI references with IRIs.

HTTP-referencable URLs are a subset of IRIs and the most commonly used identifiers in RDF statements. URLs are especially important for Linked Data, which will be described in Sec. 2.2. Common prefixes of IRIs are used to define namespaces for the resources. Different IRI namespaces are often maintained by different authorities. *Literal* values can be used for defining attributes of a resource. Literals can be strings, numbers, dates or boolean values. They are either plain (with an optional language tag) or typed. A typed literal is annotated with a datatype URI, e. g. the commonly used XML Schema datatypes. Blank Nodes represent anonymous resources which are used if an entity is only used in a local context, e. g. a relation between two entities is modeled as a Blank Node with specific attributes that specify the relation in more detail. The identifiers of Blank Nodes are only defined for the local scope of an RDF graph. Therefore, they are not unique and cannot be used in a global context.

Figure 2.1 presents an RDF example describing a publication entitled "d-complete sequences of integers" written by Paul Erdős and Mordechai Levin. Three resources, namely `dblp:ErdosL96`, `dblp:Mordechai_Levin`, and `dblp:Paul_Erdos` represent the publication and its two authors. The Dublin Core and the FOAF vocabulary are used to express attributes and relations of the document and authors.

2.1.2 RDF Graphs

The RDF model defines an RDF graph as a set of RDF triples [60]. However, the typical visualization for RDF graphs is a directed, labeled graph with Resources, Blank nodes, and Literals representing the nodes of the graph and predicates defining typed edges leading from subject to object. For example, Figure 2.2 shows the graph representation for the RDF triples in Fig. 2.1.

Definition 2.2 (RDF Graph). *A set* $\mathcal{G} \subseteq \{(s,p,o) \mid s \in I \cup B, p \in I, o \in I \cup B \cup L\}$ *is called an RDF graph.*

[1] http://www.w3.org/TR/rdf-concepts/#dfn-URI-reference

```
1  @prefix dc: <http://purl.org/dc/elements/1.1/>.
2  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
3  @prefix dblp: <http://dblp.l3s.de/d2r/resource/>.
4  @prefix dbpp: <http://dbpedia.org/property/>.
5  @prefix foaf: <http://xmlns.com/foaf/0.1/>.
6  @prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
7
8  dblp:ErdosL96 rdf:type foaf:Document ;
9      dc:title "d-complete sequences of integers" ;
10     dc:creator dblp:Paul_Erdos ;
11     dc:creator dblp:Mordechai_Levin .
12 dblp:Paul_Erdos rdf:type foaf:Person ;
13     foaf:name "Paul Erdos"@en ;
14     foaf:name "Erdős Pál"@hu ;
15     dbpp:birthDate "1913-03-26"^^xsd:date ;
16     dbpp:deathDate "1996-09-20"^^xsd:date .
17 dblp:Mordechai_Levin rdf:type foaf:Person ;
18     foaf:name "Mordechai Levin" .
```

Fig. 2.1. An RDF example which describes a publication by Paul Erdős (in Turtle notation, cf. RDF serialization formats in Sec. 2.1.4)

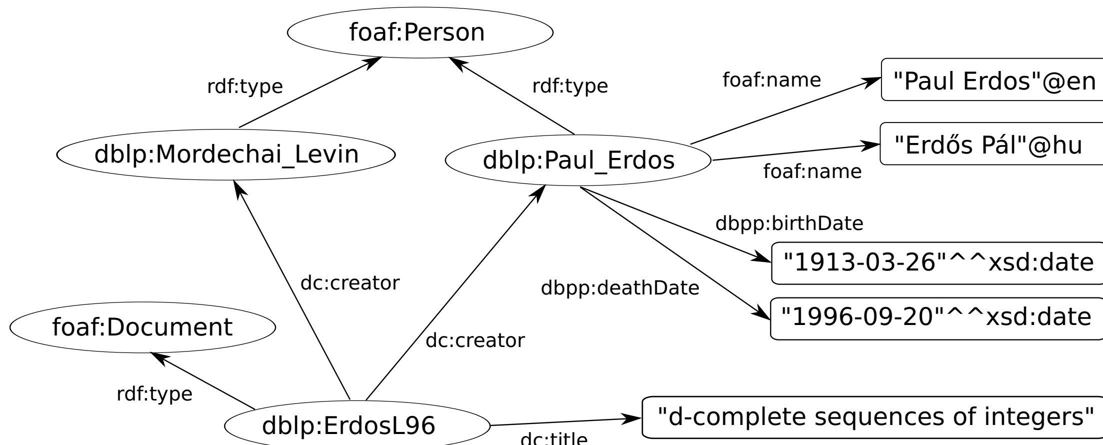


Fig. 2.2. Graph representation of the running example

2.1.3 RDF Vocabularies

The semantics of RDF statements is defined by vocabularies that are used to express resource relations and class definitions. Such vocabularies can be defined with the *RDF Vocabulary Description Language* (RDF Schema) [36] and the *Web Ontology Language* (OWL) [156, 223].

RDF Schema (RDFS) provides classes and properties for defining RDF vocabularies. Core RDF only provides `rdf:type` as property to define that an entity is an instance of a class. RDFS adds class definitions like `rdfs:Resource`, `rdfs:Class`, and `rdfs:Literal`. Moreover, the relation between entities can be specified in detail by defining `rdfs:subClassOf` or `rdfs:subPropertyOf`, and by restricting domain and range of a property with `rdfs:domain` and `rdfs:range`. An RDFS vocabulary definition itself is expressed in RDF.

Other popular vocabularies used for RDF documents are for example *Dublin Core*[2], *Friend Of A Friend*[3] (FOAF) [37], and the *Simple Knowledge Organization System* (SKOS) [122]. Dublin Core defines metadata for documents, e. g. author, title, and publication date. The Friend of a Friend vocabulary provides definitions for describing a person as `foaf:Person` including properties like name, address, and occupation. The relation `foaf:knows` can be used to build a friendship network between persons. For classification of concepts, the SKOS vocabulary defines `skos:Concept` and e. g. `skos:broader` and `skos:narrower` among others semantic relationships.

2.1.4 RDF Syntax

RDF/XML [83] is the standard serialization format of RDF. Since XML is the most popular data exchange format on the Web, the XML serialization of RDF can also be processed by many tools and applications. However, RDF/XML is not that easy to read for humans since XML is very verbose. Moreover, the tree structure of XML is not the best representation for an RDF graph. Figure 2.3 shows the RDF from the example serialized as XML.

```
1  <?xml version="1.0"?>
2  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:dbpp="http://dbpedia.org/property/"
         xmlns:dblp="http://dblp.l3s.de/d2r/resource/" >
3    <foaf:Document rdf:about="http://dblp.l3s.de/d2r/resource/ErdosL96">
4      <dc:title>d-complete sequences of integers</dc:title>
5      <dc:creator>
6        <foaf:Person rdf:about="http://dblp.l3s.de/d2r/resource/Paul_Erdos">
7          <foaf:name xml:lang="en">Paul Erdos</foaf:name>
8          <foaf:name xml:lang="hu">Erdős Pál</foaf:name>
9          <dbpp:birthDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
               1913-03-26</dbpp:birthDate>
10         <dbpp:deathDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
               1996-09-20</dbpp:deathDate>
11       </foaf:Person>
12     </dc:creator>
13     <dc:creator rdf:resource="http://dblp.l3s.de/d2r/resource/Mordechai_Levin"/>
14   </foaf:Document>
15 </rdf:RDF>
```

Fig. 2.3. RDF/XML representation of the running example

N3, Turtle, and N-Triples. The *Terse RDF Triple Language*[4] (Turtle) is a popular plain-text serialization format for RDF. It is very compact compared

[2] http://dublincore.org/
[3] http://www.foaf-project.org/
[4] http://www.w3.org/TR/2011/WD-turtle-20110809/

to RDF/XML and was designed with better human-readability in mind. Turtle allows for the definition of namespace prefixes and provides shorthand notations for consecutive triples with the same subject (c.f. Figure 2.1). Turtle is a subset of the more general *Notation 3*[5] (N3). While Turtle is restricted to RDF, N3 also includes additional features, like defining rules which is not part of RDF. *N-Triples*[6] is a simplistic subset of Turtle. Each RDF triple is written in a separate text line with fully qualified URIs. The N-Triple syntax is easy to parse for applications, but it consumes a lot of space due to its verbosity.

```
1 <http://dblp.l3s.de/d2r/resource/Paul_Erdos>
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      <http://xmlns.com/foaf/0.1/Person>.
2 <http://dblp.l3s.de/d2r/resource/Paul_Erdos> <http://xmlns.com/foaf/0.1/name> "Paul
      Erdos"@en.
3 <http://dblp.l3s.de/d2r/resource/Paul_Erdos> <http://xmlns.com/foaf/0.1/name> "Erdős
      Pál"@hu.
4 <http://dblp.l3s.de/d2r/resource/Paul_Erdos> <http://dbpedia.org/property/birthDate>
      "1913-03-26"^^<http://www.w3.org/2001/XMLSchema#date>.
5 <http://dblp.l3s.de/d2r/resource/Paul_Erdos> <http://dbpedia.org/property/deathDate>
      "1996-09-20"^^<http://www.w3.org/2001/XMLSchema#date>.
6 <http://dblp.l3s.de/d2r/resource/ErdosL96>
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      <http://xmlns.com/foaf/0.1/Document>.
7 <http://dblp.l3s.de/d2r/resource/ErdosL96> <http://purl.org/dc/elements/1.1/title>
      "d-complete sequences of integers".
8 <http://dblp.l3s.de/d2r/resource/ErdosL96> <http://purl.org/dc/elements/1.1/creator>
      <http://dblp.l3s.de/d2r/resource/Paul_Erdos>.
9 <http://dblp.l3s.de/d2r/resource/ErdosL96> <http://purl.org/dc/elements/1.1/creator>
      <http://dblp.l3s.de/d2r/resource/Mordechai_Levin>.
```

Fig. 2.4. N-Triples representation of the running example

Embedded RDF. The missing semantic structure in web documents has been one of the reasons for the development of RDF. But RDF cannot simply replace HTML pages. Usually, RDF is provided separately in addition to existing web pages. In order to combine both worlds, RDFa [112] was developed. RDFa allows for embedding RDF statements in Web documents. In contrast to Microformats, which are a pragmatic approach with limited flexibility, RDFa does support full expressiveness and features of RDF. Figure 2.5 shows the RDFa representation for a part of the running example.

2.1.5 Named Graphs

All RDF data in a file is usually interpreted as a single RDF graph. In order to define multiple RDF graphs in the same document or repository, *Named Graphs* [46] can be used.

[5] http://www.w3.org/TeamSubmission/n3/
[6] http://www.w3.org/2001/sw/RDFCore/ntriples/

```
1 <div xmlns:dc="http://purl.org/dc/elements/1.1/"
2   about="http://dblp.l3s.de/d2r/resource/ErdosL96">
3   The journal paper <span property="dc:title">d—complete sequences of integers</span>
4   was published by <span property="dc:creator">Paul Erdos</span>
5   and <span property="dc:creator">Mordechai Levin</span>.
6 </div>
```

Fig. 2.5. RDFa example with embedded RDF data

Definition 2.3 (Named Graph). *A Named Graph is a tuple* $ng = (id, \mathcal{G})$ *where* $id \in I$ *and* $\mathcal{G}$ *is an RDF graph. The functions* $name(ng) = id$ *and* $rdfgraph(ng) = \mathcal{G}$ *return the graph's identifier (IRI) and the actual RDF graph, respectively.*

Named Graphs are useful for attaching meta-data to a graph. For example, provenance information and signatures can be added as well as version or access control information. Named Graphs ensure that the meta data is separated from the graph data. A reference to the name of a named graph, i. e. the graph's URI, can be used within the same Named Graph or in other RDF graphs. Moreover, URIs may be shared between different Named Graphs but not Blank Nodes.

Named Graphs are similar to *Quads*, which add context information as a fourth element to an RDF triple. However, Named Graphs are treated as first class objects. Syntaxes for Named Graphs, which need to express the name, the graph, and the relation between them, are RDF/XML, TriX [48], and TriG. TriX is a simple XML based serialization format and TriG is a variation of Turtle, i. e. a compact plain text format, using $'\{'$ and $'\}'$ to enclose a graph.

2.2 Linked Open Data

The *Web of Data* describes a new paradigm for publishing structured data on the Web. In contrast to the *World Wide Web*, i. e. the web of hypertext documents, the Web of Data is a network of web-accessible and interlinked resources. *Linked Open Data* (LOD) [30, 111] describes machine-readable information about resources which is published on the Web according to the Linked Data Principles[7].

2.2.1 Linked Data Principles

The Linked Data Principles define a set of requirements which should be met such that the published data is useful for a wider audience. These are the Linked Data Principles as named by Tim Berners-Lee in 2006:

- Use URIs as names for things

[7] http://www.w3.org/DesignIssues/LinkedData.html

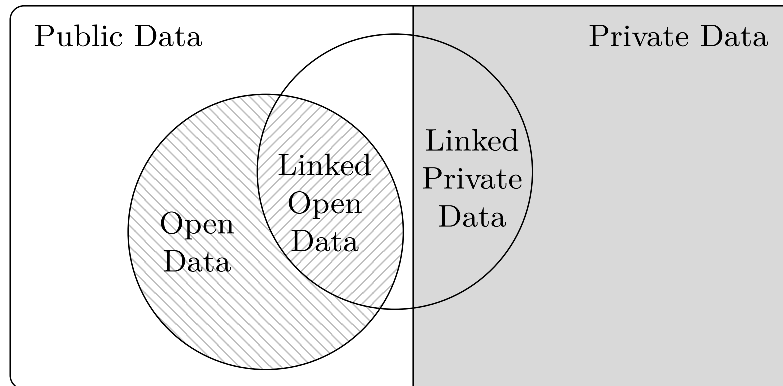


Fig. 2.6. Classification of Linked Open Data within the space of private, public and open data

- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)
- Include links to other URIs. so that they can discover more things.

The Linked Data Principles are merely guidelines. However, if not all principles are fulfilled, the re-use of data will be limited. Tim Berners-Lee says that fulfilling all four principles will offer the opportunity for unexpected re-use. The first principle is about using URIs as unique identifiers. In the semantic web every entity is identified with a URI. Principles two and three are about providing information about an entity which is identified by a URI. HTTP should be used as the standard protocol for looking up URIs and if somebody does a HTTP lookup, some useful information about the entity should be returned. Using the HTTP protocol also has the advantage that `HTTP 303` redirects can be used to refer to alternative sources for serving the description of the URI. Providing the data with standards like RDF and SPARQL allows for machine-readable information about properties, classes, and the relationship between terms. The amount of data which is considered to be useful depends on the data provider. A common practice is to return RDF triples where the URI is in the subject or object position. The fourth principle says that links to other resources should be included, so other resources can be discovered as well. The rational is that data is more meaningful if it is connected to other datasets. This allows for discovering more information than the data which is provided within a single dataset. The links in Linked Open Data are basically RDF triples where subject and object are resources from different datasets. The most common link type is `owl:sameAs`, which expresses that two entities describe the same thing. But the links can essentially have any type of link, e. g. `foaf:knows` is often used to describe relations between persons.

2.2.2 Linked Data Cloud

The number of published linked datasets has been growing rapidly in recent years. The *Linked Data Cloud* started in May 2007 with an initially small number of twelve datasets. As of November 2011, the (official) Linked Data Cloud, as visualized in Fig. 2.7, contained 295 datasets. Different topical clusters can be identified (marked with different colors). The major clusters are *Media*, *Geographic*, *Publications*, *Government*, *Life Sciences*, *Cross-Domain*, and *User-generated Content*. The most prominent dataset in the Linked Data Cloud is *DBpedia*[8]. It contains the *infobox data* of Wikipedia[9] as RDF statements. DBpedia is also the central hub of the Linked Data Cloud. It was one of the first datasets to be published as Linked Open Data. Thus, many datasets provide links to resources in DBpedia. Another reason is the good reputation of Wikipedia and the diversity of information that is provided. However, there are only a few backlinks from DBpedia to other datasets. Within the different topic clusters one can also observe that links between the datasets are sparse. Especially, mutual links are very rare. A network analysis [192] of the Linked Data Cloud in 2009 also revealed that the graph is not strongly connected. Further in-depth analysis of the Linked Data Cloud were performed by Ding et al. [68] and Bartolomeo [21].

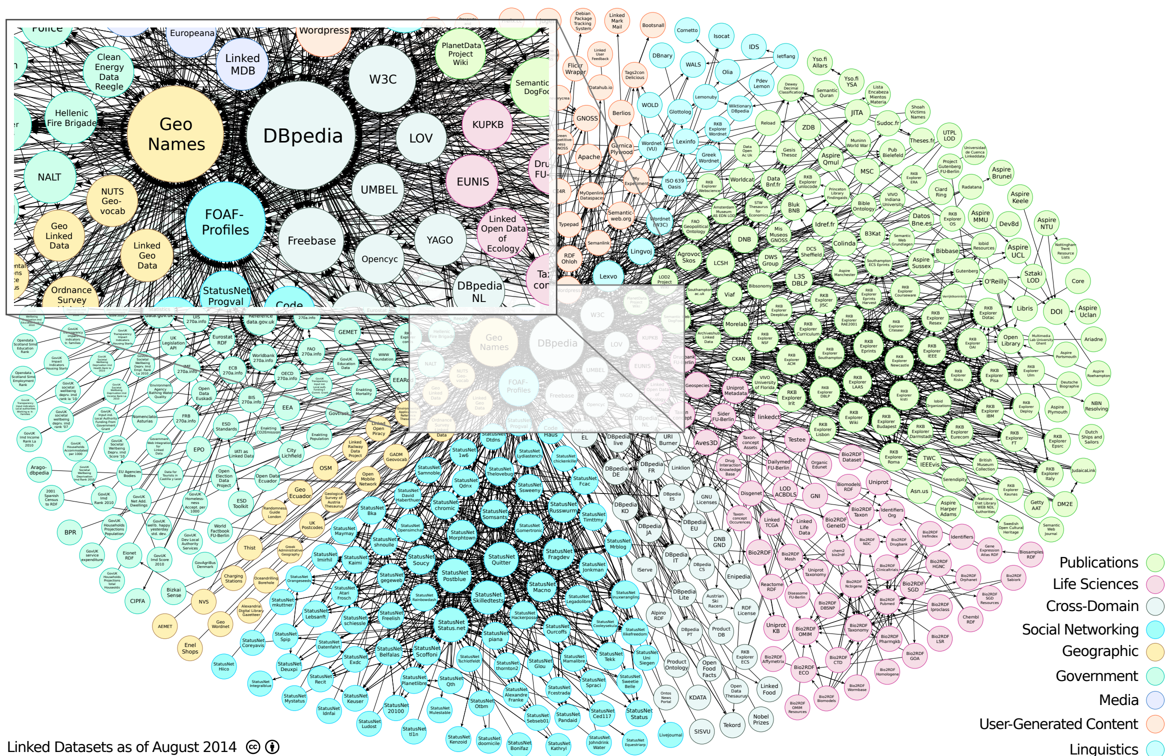


Fig. 2.7. The Linked Open Data Cloud as of August 2014, by Max Schmachtenberg, Christian Bizer, Anja Jentzsch and Richard Cyganiak. http://lod-cloud.net/.

[8] http://dbpedia.org/
[9] http://wikipedia.org

2.2.3 Consuming Linked Data

There are two different approaches for consuming linked data, i. e. keyword-based search and query processing. The second approach is similar to data retrieval in relational databases and based on the evaluation of SPARQL queries (c.f. Sec. 2.3) across the Linked Data sources. Depending on the capabilities of the data sources, there are different possible approaches for the SPARQL query evaluation, i. e. URI resolution and link traversal, integration of data dumps, or the use of SPARQL endpoints. Basically, three main query processing approaches can be considered, namely 1) *Link Traversal Based Query Evaluation*, 2) *Data Warehousing*, and 3) *Distributed Query Processing*. These three approaches will be discussed in more detail in Chapter 3. Hartig et al. [107] also give a detailed overview of the strategies and additionally discuss possible future combinations thereof.

2.2.4 Limitations of Linked Open Data

The datasets in the Linked Data Cloud are very heterogeneous and of varying quality. Some datasets fully implement all linked data principles while others support them only partially. Especially, there are datasets which are "Linked Data" or "Open Data" but not "Linked Open Data". A common problem is also the correctness and completeness of data. Datasets may contain errors, at the syntactic or semantic level. In general, common vocabularies should be used in datasets to allow for consistent interpretation of the data. Otherwise, the data may not be processed correctly. The (mis)use of `owl:sameAs` links [68, 95] is a prominent example for different semantic interpretations of the same concept. The reason is a not so clear definition of the meaning of *same as* in the Linked Data Cloud. Another problem is the sparseness of links in the Linked Data Cloud. Often there are only links from one dataset to another, but no backlinks. Consequently, there is no path between every node in the network. Hence, when exploring the Linked Data Cloud, it may not be possible to discover all relevant information. There may also be different versions of datasets which may hardly be discerned, because version information cannot (easily) be attached to URIs or RDF statements. In fact, the handling of provenance information requires special mechanisms to be implemented.

2.3 SPARQL

The *SPARQL Protocol And RDF Query Language* [187] has been developed by the *World Wide Web Consortium* and W3C and defines a standard query algebra and protocol for querying RDF data sources. SPARQL is based on graph pattern matching. A set of *triple patterns* defines a graph structure which needs to be matched by the RDF statements in the queried RDF graph. A triple pattern contains unbound variables or bound variables (i. e. constant

values) as subject, predicate, or object. SPARQL also includes further query operators like projection, filter expression, or optional parts. There are four forms of SPARQL queries, namely `SELECT`, `CONSTRUCT`, `ASK`, and `DESCRIBE` queries (c.f. Fig. 2.8). The main difference is the kind of result that is returned by each query type (cf. Fig. 2.9).

`SELECT` queries are the most common type of queries and they are comparable with SQL queries. The result is a multiset of variable bindings obtained by matching triple patterns with unbound variables against RDF statements.

`CONSTRUCT` queries are similar to `SELECT` queries. But instead of variable bindings, they return an RDF graph. The desired graph structure is defined in the `CONSTRUCT` clause and instantiated with the variable bindings obtained during pattern matching.

`ASK` queries return a boolean value as a result. For a given query pattern they return true if there is at least one valid set of variable bindings. Otherwise they return false.

`DESCRIBE` queries are used to obtain information about resources. The query result is an RDF graph which describes the resources that have been matched in the query's graph pattern. Alternatively, a `DESCRIBE` query can contain a specific URI instead of a graph pattern. The amount of information returned depends on the implementation of the query engine.

```
SELECT ?author ?coauthor
WHERE {
    ?publication dc:creator ?author, ?coauthor .
    ?author foaf:name "Paul Erdos" .
    ?coauthor foaf:name "Mordechai Levin" .
}
```

```
CONSTRUCT {
    ?author foaf:knows ?coauthor .
}
WHERE {
    ?publication dc:creator ?author, ?coauthor .
    ?author foaf:name "Paul Erdos" .
    ?coauthor foaf:name "Mordechai Levin" .
}
```

```
ASK {
    ?publication dc:creator ?author, ?coauthor .
    ?author foaf:name "Paul Erdos" .
    ?coauthor foaf:name "Mordechai Levin" .
}
```

```
DESCRIBE
    <http://dblp.de/resource/Paul_Erdos>
```

Fig. 2.8. The four query forms of SPARQL: `SELECT`, `CONSTRUCT`, `ASK`, and `DESCRIBE`

Definition 2.4 (SPARQL Graph Pattern). *Let $T = I \cup L \cup B$ be the set of all IRIs, Literals, and Blank nodes (commonly known as RDF terms [60]) and let V be an infinite set of query variables which is disjoint from T.*

Following, the algebraic definitions of [183, 200] SPARQL graph patterns are defined recursively with the binary operators UNION, AND, OPTIONAL, and FILTER[10]. A triple pattern $t \in (T \cup V) \times (I \cup V) \times (T \cup V)$ is a graph

[10] AND defines an equi-join and OPTIONAL defines a left outer join (equivalent to relational databases).

SELECT:

?author	?coauthor
dblp:Paul_Erdos	dblp:Mordechai_Levin

CONSTRUCT:

```
dblp:Paul_Erdos foaf:knows
        dblp:Mordechai_Levin
```

ASK:

```
true
```

DESCRIBE:

```
@prefix rdf: <http://www.w3.org/...>.
@prefix dblp: <http://dblp.l3s.de/...>.
@prefix dbpp: <http://dbpedia.org/...>.
@prefix foaf: <http://xmlns.com/...>.
@prefix xsd: <http://www.w3.org/...>.

dblp:Paul_Erdos rdf:type foaf:Person ;
     foaf:name "Paul Erdos"@en ;
     foaf:name "Erdős Pál"@hu ;
     dbpp:birthDate "1913−03−26"^^xsd:date ;
     dbpp:deathDate "1996−09−20"^^xsd:date .
```

Fig. 2.9. SPARQL results for `SELECT`, `CONSTRUCT`, `ASK`, and `DESCRIBE` example

pattern[11]*. If $\mathcal{P}$ and $\mathcal{P}'$ are graph patterns and E is a SPARQL filter expression, then the expressions*

$$(i)\ \ \mathcal{P}\ \texttt{AND}\ \mathcal{P}' \quad (ii)\ \ \mathcal{P}\ \texttt{UNION}\ \mathcal{P}' \quad (iii)\ \ \mathcal{P}\ \texttt{OPTIONAL}\ \mathcal{P}' \quad (iv)\ \ \mathcal{P}\ \texttt{FILTER}\ E$$

are graph patterns, too.

The SPARQL example in Fig. 2.10 selects German co-authors of Paul Erdős. The triple patterns to be matched are defined in the `WHERE` clause (lines two to twelve). Result bindings for the variables have to satisfy all triple patterns, except for the `OPTIONAL` patterns. The graph structure which is described by the triple patterns is visualized on the right side of the example in Fig. 2.10. Co-authorship is defined via the creator relation between people and articles. The German nationality is a property of a person. Line 13 defines an order on the results and line 14 restricts the number of results to 10. The first line specifies the final projection of variables. All namespaces are omitted for better readability.

```
 1 SELECT ?name ?workplace
 2 WHERE {
 3   ?author foaf:name "Paul Erdos".
 4   ?article dc:creator ?author.
 5   ?article dc:creator ?coauthor.
 6   ?article rdf:type foaf:Document.
 7   ?coauthor foaf:name ?name.
 8   ?coauthor dbpprop:nationality dbpedia:German.
 9   OPTIONAL {
10     ?coauthor dbpprop:workplaces ?workplace.
11   }
12 }
13 ORDER BY ?name
14 LIMIT 10
```

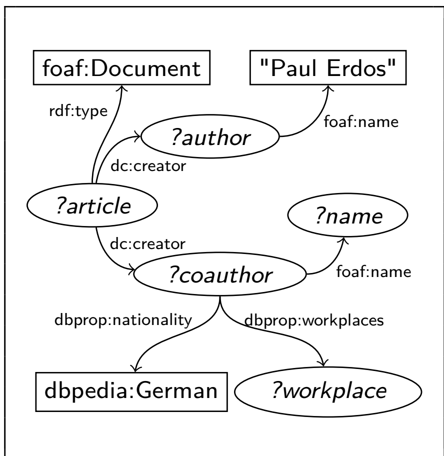


Fig. 2.10. SPARQL query example: German co-authors of Paul Erdős

[11] In contrast to RDF the SPARQL specification permits Literals in subject position.

2.3.1 SPARQL 1.1

SPARQL 1.1 [98] is a W3C proposed recommendation which includes various improvements for SPARQL. It does not only add new features to the SPARQL query language, but also extends the SPARQL protocol, e. g. with a standardized *update* mechanism for inserting RDF data into a data source. The most notable additions for the SPARQL query language are *Aggregates*, *Sub Queries*, and *Property Paths*. Aggregates apply an expression over a set of query solutions. They have been part of SQL for decades, but so far SPARQL was missing such important features. Therefore, several triple stores had already implemented their own set of aggregation functions. With SPARQL 1.1 there is finally a standard set of aggregation functions which have to be implemented, namely `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`, `GROUP_CONCAT`, and `SAMPLE`. Property paths add the ability to match paths of arbitrary length, i. e. paths between two nodes in the RDF graph which contain an arbitrary number of edges. Sub Queries are used to embed SPARQL queries in other SPARQL queries. This allows for evaluating sub expressions, e. g. in order to restrict the number of results.

2.3.2 SPARQL Federation

Federation in the semantic web context deals with the seamless integration of heterogeneous RDF data sources, like from the Linked Data cloud, such that the data which is offered by different providers is accessed, aggregated, and made available to the user through a SPARQL query interface as if it would reside in a single data source. Therefore, SPARQL federation is formally the evaluation of SPARQL queries across a federated RDF graph.

Definition 2.5 (Federated RDF Graph). *A finite set* $\mathcal{F} = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$ *with RDF graphs* $\mathcal{G}_1, \ldots, \mathcal{G}_n$ *is a federated RDF graph.*

Figure 2.11 shows some RDF statements distributed across three Linked Data sources (as illustrated in [87]). A link between two datasets exists if the same URI occurs in object position of an RDF triple in the first dataset and in subject position of an RDF triple in the second dataset.

An important addition to SPARQL 1.1 is the federation extension [186]. Two new keywords were introduced to the SPARQL algebra, namely `SERVICE` and `VALUES` (formerly `BINDINGS`). The `SERVICE` keyword allows for defining explicit SPARQL endpoints for a set of triple patterns in the query. The example in Fig. 2.12 associates the DBLP endpoint with two triple patterns. Hence, the subset will be sent to the DBLP endpoint whereas the triple pattern in line 3 will be evaluated on the default graph. The `VALUES` keyword is used for passing values for already bound variables along with the query, i. e. if a query part has already been evaluated, the resulting variable bindings can be passed on to a different SPARQL endpoint.

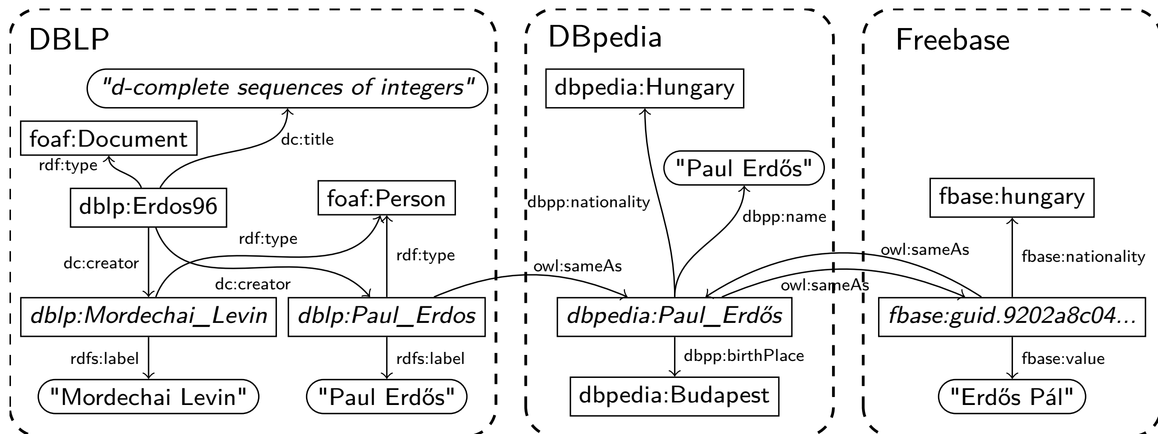


Fig. 2.11. Information about Paul Erdös in three linked data sources

Federation implementations based on SPARQL 1.0 [188, 88, 203] rely on individual mechanisms to distribute the query evaluation and merge the results in an efficient way. With the two new keywords, however, a wider range of optimizations is possible. Query optimizations for SPARQL federation will be discussed in detail in Chapter 5.

```
1 SELECT ?article ?name
2 WHERE {
3   ?author rdf:type Foaf:Person.
4   SERVICE <http://dblp.l3s.de/sparql/> {
5     ?article dc:creator ?author.
6     ?author foaf:name ?name.
7   }
8   VALUES ?author { (<http://dblp.l3s.de/d2r/resource/Paul_Erdos>) }
9 }
```

Fig. 2.12. Use of `SERVICE` and `VALUES` keywords in a federated SPARQL query

2.4 RDF Stores

There are various implementations of RDF stores which use different approaches and technologies for the persistent storage of RDF data and for efficient query processing. *Relational Database Management Systems* (RDBMS) are de-facto the first choice for storing structured data. Hence, there are several RDF store implementations which use a relational database as storage backend. On the other hand, there are also custom (native) implementations for storing RDF data.

2.4.1 RDF in RDBMS

The naive approach for putting RDF data into a relational database is to create one large table with three columns for subject, predicate, and object.

In a single table, however, all joins are self joins. Depending on the joined triple patterns, a self join involves many different areas of the table which have to be loaded beforehand. Indexes on the columns are maintained to avoid full table scans. But long seek and scan times as well as a high number of disk accesses occur if a large number of tuples, which do not all fit into memory, have to be joined.

Property Tables [229] can be used for partitioning RDF data. The motivation is that resources of the same type typically share the same properties (i. e. predicates). A property table has a column for the subject and columns for each predicate. A row contains the tuple with the subject and the object values for each predicate, thus forming a subject-object-matrix. NULL values occur if there is no RDF statement for the subject-predicate combination. There are two ways for creating property tables, either by explicit class definition (i. e. via `rdf:type`) or by clustering predicates. In the second case, a predicate must not occur in two different property tables. For predicates which cannot be assigned to any property table there is one table containing all the left-over statements using three columns for subject, predicate, and object.

The advantage of property tables is that joins over the same subject can typically be answered with the data from a single table. But there are also several drawbacks. Since RDF is usually not highly structured, there can be many NULL values in the tables, thus increasing the required space. Another drawback is that multi-valued attributes, i. e. resources with multiple values for the same predicate, cannot be represented efficiently in a property table. Class-based property tables can only be used if the queries contain `rdf:type` predicates. Otherwise they are useless. Without class information careful clustering of predicates is necessary.

2.4.2 Column Stores

Vertical partitioning for RDF was first proposed by Abadi et al. [1]. A two-column table is created for each unique predicate. The subjects of triples with the predicate and the corresponding objects are then put in the tables and sorted by subject. Compared to property tables, there is no clustering required and the tables do not contain any NULL values. The vertical partitioning approach uses a column-oriented database instead of a row-oriented database as storage backend. With column stores the data is stored in a more compact form. Especially, if dictionary encoding is used the tuples have a fixed length. Moreover, data compression techniques can be used efficiently. Table scans are much faster and, due to the sorted data, efficient merge-joins can be applied as well. For joins over objects, the tables can be sorted (indexed) by object.

Vertical partitioning is not well suited for datasets with a skewed distribution of predicates. For very popular predicates like `rdf:type` and predicates, which are not used that often, there will be a few large tables and many small tables. Additionally, if a dataset, like DBpedia, has many different predicates

there will also be a large number of tables. Vertical partitioning only works well for queries with bound predicates. Triple patterns with unbound properties have to be matched against every single table. This is very inefficient as all tables have to be scanned to match the pattern.

Figure 2.13 shows how the running example would be stored in tables for the three described approaches.

Triple Table:

s	*p*	*o*
dblp:ErdosL96	rdf:type	foaf:Document
dblp:ErdosL96	dc:title	"d-complete seq..."
dblp:ErdosL96	dc:creator	dblp:Paul_Erdos
dblp:ErdosL96	dc:creator	dblp:Mordechai_Levin
dblp:Paul_Erdos	rdf:type	foaf:Person
dblp:Paul_Erdos	foaf:name	"Paul Erdos"
dblp:Paul_Erdos	foaf:name	"Erdős Pál"
dblp:Paul_Erdos	dbpp:birthDate	"1913-03-26"
dblp:Paul_Erdos	dbpp:deathDate	"1996-09-20"
dblp:Mordechai_Levin	rdf:type	foaf:Person
dblp:Mordechai_Levin	foaf:name	"Mordechai Levin"

Property Tables:

Class: Person			
Subject	Name	BirthDate	DeathDate
dblp:Paul_Erdos	{Paul Erdos, Erdős Pál}	1913-03-26	1996-09-20
dblp:Mordechai_Levin	Mordechai Levin	NULL	NULL

Class: Document		
Subject	Title	Creator
dblp:ErdosL96	"d-complete seq..."	{dblp:Paul_Erdos, ...}

Vertical Partitioning:

Type	
dblp:ErdosL96	foaf:Document
dblp:Paul_Erdos	foaf:Person
dblp:Mordechai_Levin	foaf:Person

Name	
dblp:Paul_Erdos	"Paul Erdos"
dblp:Paul_Erdos	"Erdős Pál"
dblp:Mordechai_Levin	"Mordechai Levin"

BirthDate	
dblp:Paul_Erdos	"1913-03-26"

DeathDate	
dblp:Paul_Erdos	"1996-09-20"

Creator	
dblp:ErdosL96	dblp:Paul_Erdos
dblp:ErdosL96	dblp:Mordechai_Levin

Title	
dblp:ErdosL96	"d-complete seq..."

Fig. 2.13. for triple table, property tables, and vertical partitioning

2.4.3 Triple and Quad Stores

The triple structure of RDF statements is the reason why RDF store implementations are commonly called *triple stores*. Most triple stores also support the handling of contextual information, i. e. named graphs. In this case, an RDF statement is a four-tuple (subject, predicate, object, and context) and for a better distinction such implementations are typically named *Quad Stores*. Triple and Quad stores employ specific data structures and optimizations for efficient data storage and query answering. All URIs, Blank Nodes, and Literals in an RDF statement are basically strings of variable length. Using a string dictionary, RDF statements can be represented with fixed size tuples of integer values that point into the string dictionary. Fixed-size integer tuples and a (sorted) string dictionary are better suited for effective compression when storing the data on disk. Moreover, many operations on RDF statements (like joins) can executed more efficiently, as more more data fits into memory and the disk access is reduced.

2.4.4 Indexing RDF

Fast and efficient query processing on large datasets requires optimized indexes, i. e. additional data structures which allow for direct access to data records without scanning the whole dataset. In relational databases, indexes can be created for individual or combined table columns and their implementation is typically based on B-Trees [25] or Hash Indexes. Although RDF triple stores often employ a table-based storage layout as well, the common database indexing approaches are not well suitable for RDF graph data, e. g. because relational databases often assume uniform data distribution and value independence whereas the data in RDF graphs is highly correlated.

Index Variations

Specific indexes are required for the support of efficient triple pattern matching and fast join computation. For example, there are six possible combinations for having one or two bound variables in subject (S), predicate (P), and object (O) position of a triple pattern, i. e. S, P, O, SP, PO, and SO. If context information should be covered as well, a fourth component (C) needs to be included, thus, yielding even more index combinations. Hence, the challenge for a triple/quad store implementation is to minimize the cost for maintaining all indexes while maximizing their benefit for an efficient query execution. Consequently, there exist different approaches which have certain advantages and disadvantages. The majority of triple patterns in SPARQL queries have a bound predicate, either alone or in combination with a bound subject or a bound object, and joins are often resource-centric. Therefore, many RDF stores, e. g. [39, 230], employ a POS and SPO index alone (or POSC and SPOC index, respectively, if context information is used). Full indexing of all six index combinations, like in Hexastore [226] and RDF3X [172, 173], increases the amount of indexed data significantly, but it also allows to produce results for matched triple patterns directly from the indexes.

SPARQL queries typically contain groups of triple patterns with shared variables which define complex graph patterns, e. g. forming a star or chain structure. Therefore, an indexing approach that goes beyond single RDF statements can help to speed up join computations and, thus, improve the query processing performance. Maduko et al. [150] use pattern mining strategies in order to create subgraph summaries. This allows for matching complete graph patterns in a SPARQL query. However, since pattern mining is an expensive pre-processing step it is usually not applicable for large datasets. Tran and Ladwig [215] build a structure index using *Bisimulation* to find equivalence classes for entities with the same incoming and outgoing links. Again, the computation is expensive. Therefore, Lue et al. [146] propose a Map/Reduce-based approach to perform Bisimulation on large datasets.

Statistics and Compression

Many indexes usually also include statistical information, e. g. the number of occurrences of subject-predicate-object combinations, which is used for the cardinality estimation during the query optimization. Stocker et al. [208] extend indexes with histograms for object values in order to allow for more precise cardinality estimation of triples with bound predicate and object. Characteristic Sets [171] were proposed to further improve the accuracy of join cardinality estimation for star-shaped query patterns. The motivation is that classic selectivity estimates, based on the independence assumption, fall short for RDF due to the high correlation between predicates and objects in RDF triples and also between predicates in different RDF triples which share the same subject. Characteristic sets are basically equivalence classes for subjects with the same set of predicates. They contain counts for the number of distinct entities and the frequency of predicates within the equivalence class.

Maintaining such detailed indexes, statistics, and histograms requires sophisticated data compression to reduce the size of the stored data. RDF3X, for example, uses a sorted string dictionary that encodes the differences between consecutive strings. Atre et al. [14] use a 3-dimensional in-memory bit matrix which represents subject-predicate-object combinations with a single bit. Additionally, they apply run-length encoding compression. However, such specific compression techniques are usually only applicable for static data and cannot be used when changing datasets required regular updates. More detailed information on RDF indexing approaches can be found in [213].

2.4.5 Relational Data Integration

Most of the data that can be published as RDF is already available in relational databases. Therefore, a mapping is required which translates the content of the relational tables into an RDF graph. Tables with a single column of primary keys and more columns that define attributes for the entities identified by the primary keys can usually be mapped easily to RDF. The primary keys are interpreted as subjects, column names as predicates, and column values as objects. Special care has to be taken for foreign keys or tables with multiple primary key columns. For example, intermediate tables are used for defining n-to-m relations between different entities. They should not be mapped directly to RDF statements using the foreign key IDs. Instead, adding RDF statements for each entity, which is referenced by the foreign keys, makes more sense. Once a mapping from relational tables to RDF is defined, the data can be transformed and stored in a triple store. However, it is often desirable not to replicate the complete data, e. g. when the relational data changes frequently and the RDF representation should always be up-to-date without expensive repetitions of data imports. In that case, a wrapper like the D2RServer[12] can

[12] http://d2rq.org/d2r-server

be used. The D2RServer uses the predefined mappings to translate SPARQL to SQL queries and to automatically transform the results to RDF.

2.5 Graph Technologies

RDF is the primary data format for representing structured knowledge on the Web. However, its graph model is very flexible and can basically be used for other scenarios (outside the Semantic Web) as well. In addition, one can adopt algorithms from graph theory for an application on RDF data. Since there exist various domain-specific graph models similar to RDF it is also possible to apply graph model transformations in order to solve specific problems with graph algorithms from other domains which could not be directly applied on the RDF data. Finally, there is a growing number of graph databases for RDF which provide efficient indexing and retrieval by focusing on the graph structure of the data.

2.5.1 Graph Algorithms for RDF

The directed labeled graph that underlies the RDF data model allows for the application of a variety of graph algorithms from graph theory or other domains with graph-based data models. For example, it is possible to compute PageRank [181] on RDF entities, apply clustering techniques [91], or identify specific relationships between entities with network analysis approaches. Hayes and Gutierrez [110] propose an intermediate bipartite graph-model to allow for a direct application of standard graph libraries on RDF data. Furthermore, several graph-based approaches focus on indexing and querying RDF data, e. g. PathHop [41] deals with the reachability problem, i. e. finding a path between two entities in the RDF graph. GRIN [217], gStore [234], Maduko et al. [150], and Tran et al. [215] identify common sub graph structures, in order to build compact indexes and allow for efficient matching of graph patterns in SPARQL queries. However, the scalability of these algorithms is limited since they often have a complexity of $O(n)$ or worse (where n is the number of vertices in the graph). Hence, an application on large RDF graphs from the Linked Data cloud is usually prohibitively expensive. In order to improve the performance of complex computations on very large graph the Signal/Collect framework [212] provides means for synchronous and asynchronous execution of parallel algorithms in a distributed setup.

2.5.2 Graph Model Transformation

A graph is a versatile data structure used to model different concepts in various domains. Typical examples are knowledge representations and models for social networks or state machines. Moreover, it is possible to apply mappings across different technology spaces, e. g. between the Semantic Web and

software engineering [204]. Typical examples of graphs used in the software engineering domain are EMF[13], Ecore, MOF[14], and TGraphs [74]. They allow for defining meta models, software components, and so forth. Mappings between such graph models can be useful if the other technology space provides methods which can be employed to solve certain problems in an easier way. For example, *Ontology-driven software development* combines semantic concepts with model-based technological spaces. This allows for consistency, satisfiability, and subsumption checking in the semantic space and model transformation, merging and computation of differences in the model-based space.

Bidirectional graph transformation for combining Semantic Web and software engineering technologies is an active research topic. For example, a mapping between RDF and TGraphs has been presented in [204]. TGraphs [73] are directed graphs with typed nodes and edges. The types are stricter than in RDF and they are modeled according to a schema on a separate schema level. RDF has an implicit schema, e. g. resources used as predicates are instances of rdf:Property and resources used as objects in triples with rdf:Type predicates are instances of rdfs:Class. A transformation between RDF and TGraphs cannot preserve all semantics because restrictions apply due to the more generic RDF model. Thus, performing forward and backward transformation (RDF → TGraph → RDF) yields the original RDF graph with exception of the strictly typed model. The transformation converts the basic RDF graph structure where subjects and objects are nodes and predicates are arcs between the nodes. Restrictions on the transformation apply primarily to the schema information in RDF, i. e. the RDF graph should not explicitly contain definitions for subclasses, sub properties and domain and ranges of properties.

2.5.3 Graph Databases

Graph databases [12] ought to be an optimal choice for managing RDF graphs. In fact, RDF data can be easily stored in a graph database but the retrieval model of SPARQL is more closely related to relational databases and SQL and does not fit well for a typical graph-oriented data storage. Therefore, the execution of SPARQL queries on top of graph databases faces new challenges. For example, graph databases are designed for fast graph traversal, i. e. starting from a specific node, one can quickly discover and follow graph edges based on the defined search criteria. SPARQL, on the other hand, is used for matching arbitrary sub-graph patterns, i. e. if a triple pattern has a bound predicate with unbound subject and object variables, all graph edges of the specified type and their connected nodes have to be returned. Without any supporting index structure, graph databases have to scan the whole graph in order to compute the results for such a query. Graph database implementations which

[13] http://eclipse.org/modeling/emf/
[14] http://www.omg.org/mof/

support RDF and SPARQL, e. g. Allegrograph[15] and Neo4J[16], typically employ additional indexes in order to handle arbitrary SPARQL queries. A detailed comparison of the RDF model with graph databases is presented by Angles and Gutierrez [11].

2.6 Summary

This chapter gave a general overview of the fundamental technologies used for Web-scale semantic data management, data storage issues, and graph technologies. RDF is the core data format in the Semantic Web which allows for representing any kind of knowledge in a generic triple-based graph data structure. The Linked Data cloud is a large collection of various RDF datasets from different domains and different data providers. Links between the datasets allow for flexible mapping of similar concept and create create a large knowledge graph which is constantly growing. Data retrieval is done via HTTP URI lookups or by defining (complex) graph patterns in SPARQL queries which are evaluated on so called SPARQL endpoints.

Storage and retrieval are important aspects for managing federated RDF graphs. Hence, an overview of triple and quad stores was given. Like in relational databases the data organization, i.e. data structures and data storeage, plays an important role. Different RDF indexing approaches were presented which have certain advantages and disadvantages depending on the specific use cases. Besides the classical data integration scenario, a variety of graph-centered technologies were presented as well, e.g. including related concepts and graph-oriented databases.

[15] http://www.franz.com/agraph/allegrograph/
[16] http://neo4j.org/

3

Distributed RDF Data Source Integration

The number of RDF datasets published on the Web is rapidly growing, especially due to the efforts of the Linked Open Data initiative [30, 111]. Web-scale data integration is a major challenge in order to exploit the knowledge in these datasets. However, the heterogeneity of Linked Data, i. e. different access methods, different schemas, different sizes, and different structuredness of data sources, is problematic for a unifying data integration approach. Moreover, there are different use cases, e. g. domain-specific analysis of large and highly structured datasets (as needed by experts in the life science domain [94, 54]) or exploratory search of barely structured data across different domains in order to discover new interesting information.

Data integration has been a prominent topic in the database community for the past years. However, an adaptation for RDF and Linked Data is not straightforward due to differences in the data model and specific constraints with respect to the data access, e. g. via SPARQL endpoints [142]. Hence, there exist different integration paradigms suitable for RDF and Linked Data which are tailored for specific scenarios and requirements.

This chapter gives an overview of the diverse data integration technologies for RDF and Linked Data and highlights the respective advantages and disadvantages. The first section starts with a summary of the most important requirements for Linked Data integration in general. A short summary of related work shows that there exist different perspectives for classifying the relevant infrastructure paradigms. Then, each of these paradigms will be presented in detail with a discussion of the respective research challenges.

3.1 Linked Data Integration Scenarios

There exist different scenarios for integrating distributed RDF data. Depending on the actual use case different functionalities and infrastructure characteristics may be more important than others. In general, there are two distinguishable use cases.

Data Exploration The diversity of datasets in the Linked Data cloud confronts most users with the problem of not knowing how to find the desired information. Thus, an exploratory approach is typically used to discover interesting resources through keyword-based search and gradual refinement of the search constraint, e. g. domain, data type, or attribute ranges. Such a data exploration is typically done by non-expert users.

Data Analysis Many domain-specific scientific datasets can be found in the Linked Data cloud. They provide valuable information for users who are experts in that area. Hence, an expert user usually knows about the employed vocabularies and dependencies between different datasets. Thus, he can formulate very specific SPARQL queries in order to retrieve the desired results.

Consequently, a sophisticated data integration system should cover at least these two use case, i. e. by supporting keyword-based search and complex structured SPARQL queries.

3.1.1 System Requirements

Software engineering typically consider functional and non-functional requirements when creating the specification of a software system. Although this chapter does not deal with particular software engineering challenges for Linked Data integration, it is advisable to investigate and understand the different problem dimensions from this perspective.

Functional Requirements

The behavior of a data integration system is defined by the *functional requirements*. In general, they describe how the retrieval of relevant information works and what the expected result is.

Expressiveness There exist different means for specifying the information need of a user. First of all there is keyword-based search which matches text in RDF triples, maybe with basic filtering constraints, e. g. on types and attributes. Moreover, there are complex structured queries expressed with SPARQL used to express specific restrictions for resources with respect to the RDF graph structure and connections between different resources. In order to leverage the full potential of Linked Data, the full

expressiveness of the SPARQL query language, including conjunctive/disjunctive queries, filter expressions, and sorting, should be supported by a data integration system. Since SPARQL 1.1 there exists a specification for data updates. However, the support of data updates depends on the data provider.

Schema Mapping Linked Data sources have heterogeneous schemas, specified in explicit ontologies or defined implicitly through the use of common vocabularies. A user typically does not know about all these different schemas. Hence, queries will usually be expressed with some common ontology and need to be mapped to dataset-specific schemas in order to retrieve results.

Reasoning A key feature of the Semantic Web technology stack is the support for reasoning. It allows for deriving additional information which has not been explicitly expressed in the datasets but is defined in ontologies, e. g. subclass/superclass relations.

Precision and Recall Typical performance measures in traditional information retrieval systems are precision and recall. Precision measures if all returned results are relevant and recall measures if all relevant results were returned. Measuring precision for SPARQL queries is problematic because SPARQL has exact match semantics. Recall, however, is quite important, because ensuring result completeness may require to query many data sources in the Linked Data cloud.

Result Ranking Result sets for keyword-based search as well as SPARQL queries can be large, especially when the scope is not selective. Hence, a common practice is to present results ordered by their relevance. However, SPARQL queries do not define any sort order for the results unless it is specified explicitly. Therefore, other aspects, like authority or trustworthiness of data sources, can be used for ranking.

Flexibility Datasets in the Linked Data cloud are accessible by different means, i. e. URI resolution, data dumps, and SPARQL endpoints. The flexibility of a data integration system also depends on the support of these access methods, especially if a broad coverage of the data sources is intended. Moreover, data sources often have different capabilities, e. g. with respect to the supported version of SPARQL, additional sets of custom functions, or metadata they provide. Additionally, the underlying database implementation may include specific indexing techniques, which may be recognized in the data integration.

Non-functional Requirements

The overall qualities of a data integration system are defined with the *non-functional requirements*. This includes features like scalability, reliability, and consistency.

Scalability There is a great variety of data sources in the Linked Data cloud, ranging from a few large and highly structured scientific datasets, e. g.

UniProt [18] over multi-domain datasets like DBpedia [15, 31] to many small less structured datasets. The latter also includes fragments of larger datasets which are obtained from URI lookups and represent a dataset by their own. A scalable data integration system should be able to handle the large number of datasets as well as the size and diversity of the data. With respect to the growing number of datasets in the Linked Data cloud, including billions of triples, there is certainly the need for scaling up to data sizes which are by orders of magnitude larger in the future.

Performance Dealing with the large number of datasets in the Linked Data cloud, the variety of the RDF data and the complexity of SPARQL queries is challenging. The performance of data integration systems depends on many factors, e. g. the utilization of index structures, the cost for sending data over the network, or parallel query execution. Hence, it can be measured with different metrics, like response time, network communication overhead, or recall.
There are also extreme application scenarios with frequently updated data sources, e. g. stock exchange data or sensor data. In such cases the performance plays an important role in order to provide continuous and up-to-date results.

Reliability A Web-scale data integration has to deal with network-related issues, e. g. unreachable data sources and unpredictable network conditions (response time, bandwidth, data rate). Temporal and permanent failures of data sources can also be an issue. A failure can be extremely severe if a central component of the data integration infrastructure is affected. From the perspective of distributed systems there exist several techniques which can be applied to improve the reliability of a system, e. g. decentralization of the infrastructure as well as caching and replication of data. In the Linked Data cloud, it is already possible to find multiple copies of certain datasets and resources.

Consistency The heterogeneity of the data sources leads to certain conflicting situations. First of all, errors in datasets are quite common, i. e. syntactical and semantical errors. In the latter case datasets may violate ontology specifications which can hamper data mapping and reasoning. Moreover, there may be different versions of the same dataset available. Such conflict cannot be handled without appropriate provenance information.

Policy Enforcement Data sources in the Linked Data cloud are highly autonomous. Moreover, data providers may impose restrictions on the use of their data, i. e. there is a difference between Linked Open Data and Linked Data. In such cases it may be permitted to access the data through specific interfaces but it is forbidden to download the data and store copies at different sites. However, such restriction typically have a significant influence on the implementation of the data integration.

3.1.2 Classification of Infrastructure Paradigms

Distributed data integration has been an active research topic over the past decades, especially in the database community. Federated database systems [206] are a well-known example for integrating diverse heterogeneous data sources with different schemas. Data integration approaches in general consider two major strategies, namely *materialized data integration* and *virtual data integration*. The first defines a system where data copies are maintained in a local data store. The latter defines an infrastructure where the data is accessed on the original data sources, e. g. through a mediator [228].

With respect to materialized and virtual data integration, it is possible to differentiate between three general feature dimensions, namely *autonomous/cooperative data sources*, *central/distributed data storage*, and *central/distributed indexing* [87]. Specific feature combinations, as depicted in Fig. 3.1, characterize three different infrastructure paradigms, i. e. *data warehouse*, *federation*, and *Peer-to-Peer data management*.

<table>
<tr><td></td><td colspan="2">Central Data Storage</td><td colspan="2">Distributed Data Storage</td></tr>
<tr><td>Autonomous Data Sources</td><td>n/a</td><td rowspan="2">Data Warehouse</td><td rowspan="2">Federation</td><td>n/a</td></tr>
<tr><td>Cooperative Data Sources</td><td>n/a</td><td>P2P Data Management</td></tr>
<tr><td></td><td>Distr. Index</td><td colspan="2">Central Index</td><td>Distr. Index</td></tr>
</table>

Fig. 3.1. Data integration characteristics (see also [87]) which yield different infrastructure paradigms

This classification is also applicable for RDF data integration in general. But there are more aspects to be considered, especially with respect to the characteristics of Linked Data and the functional and non-functional requirements mentioned before. Hence, it is not surprising that there exist different perspectives on Linked Data integration in the research community.

Hartig and Langegger [107] take a "database perspective on consuming Linked Data". They classify five different query processing approaches with respect to several properties (functional and non-functional), namely *universe of discourse*, *source access*, *use of original data*, *supporting data structures*, *response time and throughput*, *precision and recall*, and *up-to-dateness*. Their considered architecture paradigms are *data warehousing*, *search engines*, *query federation*, *active discovery query federation*, and *link traversal*.

Hausenblas and Karnstedt [109] argue that Linked Open Data can be understood as a "Web-scale database". They compare (mostly technical) features of RDBMS with Linked Data, like *operations and language*, *catalog*, *user views*, *integrity*, *data independence*, and *distribution independence*, and highlight re-

quirements for a "Linked Data Base", e. g. *schema integration*, *ranking*, and *provenance*. Their view on integration paradigms includes *centralized repositories*, *live look-ups*, and *peer-to-peer systems*.

A similar perspective is taken by Hose et al. [116] with focus on centralized and distributed data management. They distinguish between *data warehouse*, *search engines*, *federation*, *link traversal*, and *peer-to-peer systems*. In addition, they also consider reasoning on uncertain RDF databases.

Ladwig and Tran [135] focus on linked data query processing and consider three challenges: (1) the *volume of the source collection*, i. e. dereferenceable URIs are considered as distinct data sources, (2) the *dynamics of the source collection*, e. g. sources with (frequently) changing content, and (3) the *heterogeneity*, *access options*, and *descriptions of sources*. Based on these aspects they differentiate between a *top-down*, a *bottom-up*, and a *mixed* query processing strategy, which basically covers *federation* and *link traversal* as infrastructure paradigms.

There is obviously a common agreement on the distinction of the major infrastructure paradigms in the community. Table 3.1 is an attempt to classify and compare these paradigms based on selected characteristics. But there exist also hybrid approaches, e. g. federation combined with link traversal [135], which combine certain characteristics. In addition, the table includes a sixth paradigm which has not received much attention yet, i. e. the publish/subscribe paradigm. However, due to recently increased interest in the application of RDF data in event-based systems, there is probably more research activity in this area to be expected. Therefore, this overview is merely a summarization of the current state-of-the-art and may not be complete. New data integration strategies may be developed in the future.

The first part of the comparison distinguishes between local and distributed data/index management. It is complemented with a choice of four non-functional characteristics and a classification ranging from poor to high which reflects the relative perceived quality. A detailed discussion of each of the paradigms is presented in the sections below.

Table 3.1. Comparison of infrastructure paradigms

	Data Warehouse	Search Engine	Federation	Link Traversal	Peer-to-Peer	Event-based
integration	material.	virtual	virtual	virtual	material.	virtual
data	local	distributed	distributed	distributed	distributed	distributed
index	local	local	local	—	distributed	—
scalability	medium	medium	good	good	high	good
reliability	good	good	medium	medium	good	good
consistency	poor	medium	good	good	poor	good
performance	high	high	medium	poor-medium	medium-good	medium

3.2 Data Warehousing

A data warehouse [52] implements materialized data integration by storing data copies from different data sources in a centralized database. No network communication is required for answering queries on the data. Moreover, optimized data storage and indexing is applied which allows for very efficient query processing. Storing RDF in a data warehouse is recommendable if large datasets need to be analyzed quickly with complex SPARQL queries.

A data warehouse is typically used to capture a historic view on the data and allows for applying sophisticated data analysis methods, like *Online Analytical Processing* (OLAP) [56]. OLAP employs a multidimensional data model, i. e. a data cube, with hierarchical levels containing statistical facts. Functions like *selection*, *projection*, *drill-down/roll-up*, and *slice/dice* are used to aggregate and analyze information from the dataset. Certain datasets from the Linked Data cloud may also be analyzed in this way, but using OLAP makes most sense when it is applied on multi-dimensional statistical data. In fact, there are a few approaches [76, 124] which aim for implementing OLAP-like analysis features for RDF data.

3.2.1 Materializing Linked Data

Data Warehouses use the *Extract-Transform-Load* (ETL) process to integrate datasets in their local database. Extraction of Linked Data works best if data dumps are available. Otherwise, URI lookups or queries on SPARQL endpoints can be used to crawl the data. In order to keep track of the origin of the data, Named Graphs [47] can be used. Transforming and loading the data into a data warehouse is typically a time consuming task because index building and the creation of statistical information is computationally expensive.

Linked Data is very diverse and datasets can change from time to time. Updating a data warehouse can require repeating the whole ETL process because indexes and statistics may need to be rebuild. Ideally, a data warehouse should be set up once and not modified afterwards, or, in the case of Linked Data, it only suitable for datasets which do not change.

3.2.2 Data Clustering

Large RDF datasets require highly scalable and fast triple store implementations. Centralized data warehouse solutions offer high performance but they cannot easily scale up to integrate all datasets from the Linked Data cloud. A scalable architecture is necessary in order to cope with current and future dataset sizes. Hence, some triple store implementations, like Virtuoso [75], 4store [97], and YARS2 [102], support clustering of RDF data.

Data clustering distributes all data across different independent storage nodes. A central master node takes care of the data organization and controls the query evaluation. Clustering provides a high scalability as storage nodes

can be easily added and removed. Special attention has to be payed to the distribution scheme of the underlying data structures. The partitioning of relational tables, column stores [1], native implementations as property tables, and the supporting indexes pose its own challenges. In general, the partitioning should take the RDF graph structure into account and allow for executing arbitrary SPARQL graph patterns efficiently.

Recent research on RDF clustering has been focusing on using distributed key-value stores [193, 118, 232], like Google's BigTable [51], Amazon's Dynamo [62], Cassandra [137], and Apache Hadoop [227], which allow for fast parallel processing of huge amounts of data based on the *MapReduce* [61] paradigm. An application on Linked Data [119] requires to map the RDF graph structure to the key/value model, e. g. using a data column layout similar to property tables [229]. However, MapReduce implementations do not support join operation which is a problem for complex queries with conjunctions of triple patterns. Thus, the main challenge is to find optimal data placement strategies which allow for matching a complex graph structure at a single storage node, without explicit join computation. In addition, the objective of MapReduce-based approaches is to support massive parallel execution for a fast query evaluation.

3.3 Search Engines

The Search for RDF data, e. g. in the Linked Data cloud, is not so different from traditional keyword-based search on the Web. Semantic Search Engines, like Swoogle [67], Sindice [178], Watson [72], SWSE [100], and Falcons [53], crawl the Web for RDF documents, extracted relevant metadata, and store it in local indexes. A search for specific keywords is performed as lookups on the indexes in order to return relevant documents, entities, or RDF triples.

Search engines are suitable for exploring the Linked Data cloud and for finding descriptions for specific resources. However, the mentioned search engines offer different search features and various result presentations. A detailed overview of the individual search engines is given by Hose et al. [116]. Following is a summarization of commonly applied techniques with respect to crawling, indexing, and ranked retrieval. Table 3.2 compares the most important features of RDF-based search engines.

3.3.1 Crawling and Indexing

Documents containing structured data can be found on the Web in different formats, e. g. RDF/XML, NTriples, embedded in Web pages as RDFa or even as Microformats. Besides, RDF data is accessible as large data dumps, through SPARQL endpoints, or as the results of Linked Data URI lookups. The crawler of a search engine automatically discovers RDF documents with the help of traditional Web search or by following links of previously crawled documents.

Table 3.2. Comparison of Semantic Search Engines (the year refers to the relevant scientific publication)

	Swoogle (2004)	SWSE (2007)	Watson (2007)	Sindice (2008)	Falcons (2009)
Focus	Documents	Entities	Ontology documents	documents, SPARQL endp.	Entities
Discovery	Web Search, semantic relations	Links	Web Search, Swoogle, Links	Links	Links
Index	N-Grams, URIs	objects, triples, joins	keywords	keywords, URIs, property:value	keywords, classes
Results	document links + metadata	entities with description (type, abstract, image)	documents/ entities	documents	entities with description and links
Ranking	PageRank on documents	PageRank on documents and RDF graph	–	TF-IDF on documents	term-based similarity and popularity
Filter/ Refine	metadata (language, type, ...)	–	classes, properties, entities	property, type, ontology, format	class/type
Features	advanced keyword queries	entity consolidation	advanced keyword queries	SPARQL, triple preview	class-inclusion reasoning
URL	http://swoogle.umbc.edu	http://swse.deri.org	http://watson.kmi.open.ac.uk	http://sindice.com	http://ws.nju.edu.cn/falcons

The document analysis includes the extraction of keywords from literals and URIs, collection of entities and their attributes, and the identification of links between documents and resources. SWSE [100] also applies entity consolidation in order to identify different references which refer to the same real world entities.

The mapping of keywords to entities and documents is commonly done with inverted indexes. A typical approaches is to employ full-text indexing, e. g. with Apache Lucene[1]. Some search engines [100, 178] also index the triples in order to allow for more complex queries, like in a data warehouse. Crawling and indexing is a continuous process as new data is constantly added and updated in the Linked Data cloud.

3.3.2 Ranked Retrieval

The search engines can be divided into *document-centric* and *entity-centric* systems, i. e. the result set contains either documents or entities for the respective search request. Due to the graph structure of RDF the scope of a keyword-based search can be restricted to specific objects, i. e. literals, URIs,

[1] http://lucene.apache.org/

properties, or classes. Additionally, result documents can be filtered by certain attributes. Falcons [53] also provides reasoning for class inclusion.

Similar to traditional information retrieval, the search results are commonly ranked by relevance. The document-centric search approaches [67, 178] apply e. g. TF-IDF [17] based scores and PageRank [181] on the document network while the entity-centric search engines compute PageRank on the RDF graph [100] or determine the relevance based on term similarity and popularity [53]. The presentation of the results differs significantly. Swoogle just returns document links with basic metadata. Sindice shows additional information about the documents and gives a triple preview. Watson provides document and entity links. SWSE and Falcons show entities with structured metadata such as type, properties, links, and images.

3.4 Federated Systems

A materialized data integration for Linked Data has several drawbacks due to the use of local data copies (cf. Sec. 3.2). The virtual data integration executes live queries on the actual data source. Hence, it returns up-to-date results and can also integrate data sources which prohibit copying their data. Because of this flexibility, the federation approach is usually better suited for scalable Linked Data integration than the data warehouse approach. The implementation of Linked Data federation is commonly based on a mediator architecture [228] and the most important components are the data source management and the distributed query optimization. However, there exist many different approaches for RDF federation [190]. Figure 3.3 shows a comparison with respect to the most common features.

3.4.1 Mediator-based Architecture

A mediator is a central node which provides the transparent integration of different data sources, i. e. it exposes a common query interface to the user which can be used as if all the data would be available from a local database. In fact, the distributed data sources do not need to be aware of the federation. Besides slight variations, the architecture of a mediator-based Linked Data federation basically look the same as for a federated database. An example for a mediator-based architecture is shown in Fig. 3.2. It depicts the components of SPLENDID, the federator implementation which will be explained in more detail in Chapters 4 and 5.

A federated database integrates different data sources through wrappers which implement schema mapping and provide (statistical) information about the data sources. A Linked Data mediator has no need for wrappers because RDF and SPARQL already provide a common data format and query interface. In order to manage the set of federated data sources, metadata and statistical information is stored in a local catalog. In the case of SPLENDID, this information is obtained through VOID descriptions.

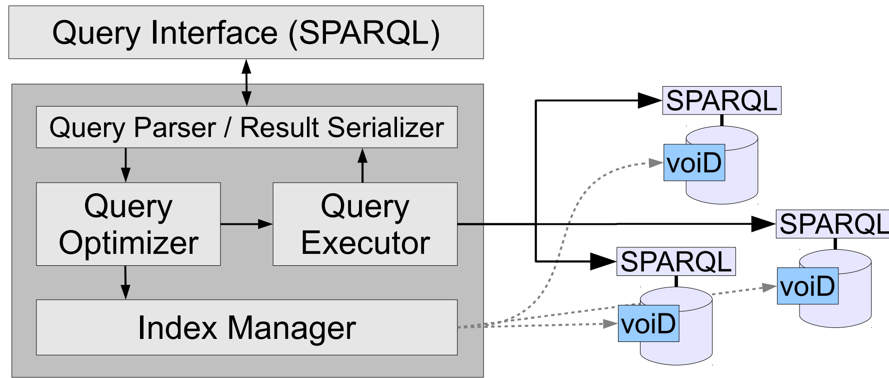


Fig. 3.2. Overview of the SPLENDID federation architecture

The query execution pipeline is the main part of a mediator. It starts with submitting SPARQL queries to the SPARQL interface and a query parser which translates the query into a logical query plan. A pre-processing step applies certain query transformations like schema mapping. Then follows the distributed query processing with source selection, query optimization and query execution. The last part is the post-processing of the obtained result set, e. g. duplicate removal, and the serialization into the desired output format.

3.4.2 Data Source Management

The responsibility of the data source management component is to organize the integrated datasets and maintain a data catalog with the metadata and statistics which is needed for source selection and query optimization.

Source Discovery

There are two different ways for integrating different RDF data sources in a federation, i. e. static setup or dynamic discovery. The first requires explicit (manual) definition of the known data sources. It is suitable for scenarios which integrate only a few large datasets, as in the Life Science domain [94, 26, 54, 222, 13, 166]. Dynamic discovery is more flexible but requires suitable discovery services for on the fly identification of relevant (linked) data sources, e. g. search engines like Sindice [178] can return a list of relevant datasets for given RDF terms. But information about SPARQL endpoints may be missing. Moreover, frequent source discovery requests can significantly hamper the performance of the query processing.

The *CKAN data hub*[2] is a recent effort to provide a directory with information about more than 4000 linked datasets. It is useful to get an overview about existing data sources and to obtain basic metadata about the datasets, e. g. employed vocabularies and the URLs of SPARQL endpoints. However,

[2] http://thedatahub.org

a federation infrastructure also needs schema information and certain statistical data from the datasets in order to perform efficient source selection and query optimization. Hence, setting up a federated system with (initial) datasets typically requires a pre-processing step for collecting such metadata and statistics. But most Linked Data sources offer only limited or no information at all about their data and the CKAN repository has seldomly complete information at the required granularity. A common format for providing such information has recently been proposed with the *Vocabulary for Interlinked Datasets* (VOID) [9] which allows for describing Linked Data with metadata and basic statistics. Although it has not been accepted as a standard yet, there is a growing number of data providers which support VOID.

Data Catalog

The data catalog contains all the information which is required for schema mapping, source selection, and query optimization. Relevant metadata about data sources include the SPARQL endpoint, used ontologies and vocabularies, extracted schema data, mapping rules, and links to other data sources. Therefore, the data catalog employs different data structures to maintain this information. Inverted indexes are needed by the source selection in order to map RDF terms to data sources. Additionally, a cost based query optimization requires statistical information about the data sets, e. g. frequency counts for resources, predicates, and correlated items. Statistics can also be used for data source ranking as in [135].

The general challenge for effective data source management is to find the optimal trade-off between the benefits of having sophisticated statistics and the cost for maintaining the data catalog. More detailed statistics will basically allow for better source selection and query optimization. But managing a large data catalog can be prohibitively expensive (in terms of size and update costs). Moreover, a data catalog should ideally fit into main memory in order to avoid expensive I/O operations. Compression techniques can effectively reduce the size of the data catalog. But existing implementations for efficient in memory storage of data statistics [14] do not consider distributed scenarios yet.

Schema Index Most SPARQL queries contain triple patterns with schema-related constraints, i. e. bound values for predicates or class types. A schema index is useful to map such triple patterns to data sources. DARQ [188], SemWIQ [140] and SPLENDID [88] maintain schema indexes with information about the number of occurrences of each predicate and type in a dataset. Such an index is very space efficient since the number of predicates and types in a dataset is limited. However, basic count-based indexes do not capture any correlation in the data.

Structure Index One of the first RDF federation systems used predicate-based path indexes [211, 7] in order to allow for efficient matching of path queries on a number of data sources. Such structural information can be

very helpful for matching SPARQL graph patterns effectively. In order to capture arbitrary graph patterns it is necessary to identify all common sub graphs in the RDF datasets. However, such an analysis, e. g. as done by Maduko et al. [150] and by Tran and Ladwig [215], is usually expensive and has to be done as an off-line computation. Therefore, it is not suitable for datasets which are updated frequently.

Instance Index Schema information is often not sufficient for effective source selection and query optimization because SPARQL queries can also contain instance data. But storing information for all URIs and literals of the federated data sources in the data catalog is impossible. This would basically lead to a complete replication of the federated datasets. Hence, instance-level data can only be maintained in a highly compressed form. A common approach in databases is to employ histograms for attributes. But the application of typical histograms for RDF data is problematic due to incomplete schema information and data correlations. An interesting approach has been presented with QTrees [101] which combine a tree structure with histograms in order to store triple pattern combinations with statistical informations for source selection and query optimization across many Linked Data sources. However, the initial QTree generation is expensive and updates of the tree structure are hardly possible later-on.

Creating the data catalog for a federation can be expensive. For example, DARQ [188] relies on hand-crafted capability files for each data source. Other approaches like SPLENDID [88] and WoDQA [8] rely on VOID descriptions. However, if the required metadata is not available, it has to be obtained by other means, e. g. through automatic extraction from data dumps [34] or by analyzing RDF data crawled from URIs and SPARQL endpoint [131].

3.4.3 Distributed Query Processing

The main task of a federator is the *distributed query processing*, i. e. implementing a query processing pipeline with *source selection*, *query optimization*, and *query execution*. All three parts have significant influence on the efficient query execution.

Source Selection

The source selection determines a mapping from SPARQL graph patterns to data sources which are expected to return results for theses patterns. Depending on the source selection strategy this information can be obtained from stored mappings in the data catalog or through live lookup via a registry service. Although live lookups impose extra cost for every processed query they also enable the retrieval of results from previously unknown sources. A local index is limited in size and often contains only schema-level information. This is usually problematic for less selective query patterns, e. g. rdfs:label, which

match most of the data sources. Hence, reducing the number of selected data sources is important, as requests to data sources which cannot return any results decreases the query execution performance. While additional statistics can be helpful it is also possible to rank selected data sources based on their relevance for the query, e. g. [135] includes only the top-k data sources for query processing.

Query Optimization

The main goal for the query optimization is typically to determine the fasted query execution plan. But in a distributed setting there are also other important aspects like the cost for network communication and the reliability of the queried data sources. There exist different optimization strategies, mostly with focus on the *join order optimization* because the size of intermediate result sets has the most influence on the cost for query processing and network communication.

Static query optimization (also referred to as optimize-then-execute approach) tries to find the best query execution plan using heuristics (FedX [203]) or cost-based query optimization (DARQ [188], SPLENDID [88]) and executes all query operators in the specified order. Adaptive query optimization takes into account that network conditions are not predictable and may change unexpectedly, even during the execution of a query. Thus, a query plan which was previously considered optimal may become inefficient. An adaptive optimization approach can cope with such changes through altering the query execution plan dynamically (Anapsid [6]) or by using different execution plans in parallel (Avalanche [22]).

Query Execution

The query execution is tightly coupled with the query optimization. On the one hand it has to execute the query plan which has been generated by the optimizer. On the other hand it restricts the optimization space to the supported physical query operators. In fact, there exist different physical join implementations, like SHI-Join [136] or Controlled Bind Join [203] which are more efficient in a distributed system than a nested-loop join or sort-merge join. Moreover, queries to different data sources can be sent in parallel. Thus, result tuples from different input streams can be processed simultaneously. There are two approaches for processing a query execution plan, i. e. the *iterator-based evaluation* (pulls result tuples) and the *data driven evaluation* (pushes results tuples). Finally, the post-processing ensures that all SPARQL result modifiers, like `DISTINCT`, are applied before the result tuples are serialized in the expected output format.

Table 3.3. Overview of federation systems

	catalog	source selection	query optimization	query execution	remarks
FedX [203]	cached ASK query results	explicit list, ASK queries	heuristics	bind join	
SPLENDID [88]	VOID	explicit list, schema data, ASK queries	cost-based (dyn. programming)	bind join, hash join	
DARQ [188]	explicit capabilities, counts and selectivity	schema index	cost-based (dyn. programming)	nested-loop join, bind join	limited to bound properties
SemWIQ [140]	schema-level, classes (monitored, aggregate queries)	subject clustering and type mapping	—	nested-loop + hashed bind join	bound types
Stuckenschmidt et al.[210]	schema-level path index	longest path	cost-based (simulated annealing)	nested-loop join, hash join	only path queries
Networked Graphs [196]	—	explicitly defined	heuristics	semi-join variation	declarative SPARQL view extension
Data Summaries [101]	Q-Tree with triples put in hash buckets	Q-Tree lookup, join overlap	—	(in memory)	source selection for joins
SPARQL-DQP [40]	—	explicit endpoints defined	RDBMS optimization techniques	unknown	based on OGSA-DQP [148]
Anapsid [6]	list of sources with concepts and capabilities	based on estimated execution time	adaptive bushy plans, small sized sub queries	adaptive XJoin, dependent join	
Avalanche [22]	schema index	directory search, split by molecules	using objective function	shipping data, local join	parallelized pipeline, bloom filters
FeDeRate [54]	—	explicit endpoint specification	heuristics/ rules	(unknown)	queries based on shared ontology
DisMed [166]	schema-level data, properties and classes (monitored)	subject-clustered, checks all sources for capabilities	basic heuristics (filter push-down)	nested-loop join, bind join	
Min-Tree BGP [221, 225]	schema-level	predicate-based	cost-based, minimum spanning tree	bind join	
Prasser et al. [185]	PARTTree, extension of QTree with type, vert. partitioning	compressed schema + instance	cost-based, top-down iteration, operator pruning	(depends on local database)	intermediate results are loaded in local database

3.5 Link Traversal Query Processing

Due to the heterogeneity and the constant growth of the Linked Data cloud it is impossible to know all data sources in advance. However, the evaluation of complex SPARQL queries should include all Linked Data sources which can contribute results. Hence, the objective of the *link traversal query processing* [104, 105] is to discover new data sources on the fly through URI lookups while executing a query. In this sense, link traversal query processing can be seen as a specialization of federation [136], solely based on datasets which are published according to the Linked Data principles (cf. Sec. 2.2).

The basic query evaluation strategy works like this [104]: (1) start with a triple pattern which contains a resource URIs, then (2) resolve the URI and store the retrieved *descriptor object*, i. e. RDF triples describing the resource, in a local dataset and (3) match the triple pattern against the *descriptor object* to create an intermediate result set for the respective variable bindings. Then, (4) continue with a triple pattern that can be joined via the variables and iteratively repeat steps (2) and (3) for newly discovered resource URIs in the intermediate result set. The solutions for all variables in a query are complete if the subset of RDF triples used for the evaluation contains all reachable URIs and if all query patterns are applied on this subset to produce the result bindings. An URI is reachable if there exists a path in the Linked Data graph which starts at an URI contained in the query patterns and is connected by RDF triples that can be matched by the query's triple patterns.

An advantage of the link traversal approach is that it requires *zero knowledge* [103], i. e. there is no need for initial dataset specifications or statistical information. All information used for query planning and evaluation is contained in a SPARQL query and retrieved on the fly through URI resolution. However, pre-collected data, e. g. from previously executed queries, can improve the source discovery and, therefore, also the result completeness. The basic requirement is that SPARQL queries always contain resource URIs which can be resolved to retrieve a set of suitable RDF triples describing the resource. Table 3.4 gives an overview of the variations of link traversal approaches.

3.5.1 Non-blocking Query Execution

The basic link traversal query processing [104] is implemented as a pipeline with nested iterators (left-deep tree). Each iterator fetches result bindings from its predecessor, performs URI resolution for newly discovered resources, and matches the obtained data against the triple pattern to generate result bindings for its ancestor. The performance of the query execution is dominated by the latency of the URI resolution, i. e. the whole pipeline can be blocked by a single iterator if a server responds slowly or due to network issues. The basic solution for this problem, as presented in [104], is prefetching of URIs and *non-blocking iterators* which can reject bindings for later processing.

Although, the iterator-based approach is very common for SPARQL query evaluation, its *pull-based* strategy is not really suitable for processing parallel requests to many Linked Data sources. Therefore, Ladwig and Tran [135, 136] propose a *push-based* strategy for query operator evaluation. In fact, the distinction between *iterator-based* and the *data-driven* query evaluation is well known in the area of adaptive query optimization in databases [65]. In the data-driven (push-based) approach each operator forwards its results binding to its successor. Thus a streaming-based binding propagation is implemented and results are returned as soon as they become available. This also has the effect that the first results are produced much faster than in the iterated based approach, which also has a certain processing overhead due to rejection and re-evaluation of bindings. However, the time for retrieving the complete result set is basically the same in both approaches as it depends only on the time which is required to receive all data from the data sources.

Another variation for event-driven query evaluation is presented by Miranker et al. [157] with their Diamond query engine. It is based on the *Rete* match making approach [80] which is commonly applied for rule-based systems. A further difference of the push-based approach by Ladwig and Tran [135, 136] is the inclusion of locally stored data. It realizes a hybrid federation approach with a special query operator, i. e. the Symmetric (Index) Hash Join, which also fixes a problem with the iterator-based implementation. The execution order of an iterator pipeline can produce incomplete results if data is processed by one iterator and removed before another iterator can retrieve additional data based on that information. The push-based approach ensures that all reachable URIs are being considered during the query evaluation.

3.5.2 Limitations of Pure Linked Data Queries

The link traversal query processing is very flexible and scalable because it does not need any kind of metadata or statistical information for source discovery. However, it also comes with several limitations, as explained below.

Results completeness The link traversal is limited to finding resources which are reachable from the starting URIs via graph edges matched by the query patterns. Incomplete results my be returned if the URI lookup does not return all relevant information[3], e. g. for inverse relations or if resource URIs are reused in other namespaces. The first case is quite common because information about links is typically stored at the source and not at the destination of the link. The latter case refers to third-party resource descriptions which reuse URIs directly (instead of defining `owl:sameAs` links).

[3] According to the Linked Data principles an URI lookup should return “interesting data”, but the interpretation is up to the data provider. Common practice is to return RDF triples which contain the resource URI in subject or object position.

Query Expressiveness SPARQL queries have to have at least one query pattern with a bound instance URI to define the starting point for the link traversal. However, URIs in object position are generally problematic because inverse relations cannot be discovered easily. Moreover, generic queries which contain only schema descriptions or literals cannot be processed. In addition, automatic schema mapping is not supported and lightweight reasoning has just recently been proposed [218]. Hence, the user basically has to know the vocabulary and how links connect the datasets to formulate a query.

Communication Cost The query processing may need to resolve a potentially large number of URIs, e. g. because of resources with many links. Thus, query performance can be hampered due to the overhead for doing many lookup and high latency of servers which need to handle many individual URI lookups. A possible solution for reducing the number of requests is source ranking [135].

Query Optimization The execution order of query operators has a significant influence on the performance. However, traditional query optimization strategies from relational databases are not applicable for pure link traversal query processing because it is based on “zero knowledge” [103]. Thus, it is only possible to rely on heuristics, e. g. which take into account the query structure and selectivity of RDF terms. Hybrid approaches [135, 136], which integrate indexed data, have certainly more potential for applying effective query optimization techniques.

A main argument for link traversal query processing is the limitation of predefined datasets in typical federation approaches. With respect to the restrictions discussed above, it seems that only a hybrid approach can provide the desired flexibility and also allows for answering complex SPARQL queries in an efficient way.

Table 3.4. Overview of link traversal approaches

	recall	query optimization	query execution
Iterator-based Link Traversal [104, 105]	incomplete results	—	iterator
Symmetric (Index) Hash Join [135, 136]	all reachable URIs	with locally indexed data	data-driven
Reasoning enabled Link Traversal [218]	extended result set	—	iterator
Diamond query engine [157]	all reachable URIs	—	Rete

3.6 Peer-to-Peer Systems

Peer-to-Peer (P2P) systems have been designed to overcome scalability and reliability issues of centralized data integration approaches, like data warehouses. They provide decentralized data storage and fault tolerance in a network of autonomous peers. With the increasing number of large RDF datasets the application of Peer-to-Peer systems for RDF data integration has been receiving more attention. The research in this area focuses mainly on data distribution and efficient query processing. But before discussing different aspects of Semantic Peer-to-Peer systems [207] it is necessary to briefly summarize typical Peer-to-Peer network topologies. A detailed survey of peer-to-peer content distribution can be found in [10]

3.6.1 Overlay Network Topologies

Peer-to-Peer systems abstract from the physical network structure by employing a logical *overlay network* to connect the participating nodes. Depending on the application scenario, there exist three main topologies for organizing the overlay, namely *unstructured*, *super-peer based*, and *structured* networks.

Unstructured Networks integrate different peers in the most flexible way. There is no specific structure defining how peers are connected. Unstructured Peer-to-Peer systems are typically document-oriented. Algorithms can establish connections between peers based on different criteria, like similar content. Each node maintains a list of its neighbors. Requests are distributed to all nodes in the network using *flooding*. If a node can answer a request with resources from its local database, a response is returned to the node which has sent the query. Flooding greatly limits the scalability of the peer-to-peer network due to the exponentially increasing number of messages. Sending requests with a timeout can prevent flooding of the whole network, but there is no guarantee for finding the desired data within the fixed maximum number of routing hops. Each peer can also keep information about the content of its neighbors in order to select suitable peers when forwarding a request. Semantic clustering [175] or semantic routing strategies [214, 145] can improve the connections between peers with similar content. An example are *Semantic Overlay Networks* (SON) [57] which create multiple overlay networks for different topics of a topic hierarchy.

Super-Peer Networks solve the scalability limitations of unstructured networks by introducing a separate layer of *super-peers* which are responsible for specific tasks like indexing, query routing, and semantic mapping [170]. Other peers are connected to these super-peers, e. g. in a star-like topology. Edutella [169] is a prominent Peer-to-Peer system which employs super-peers. The super-peers are connected with the so called *HyperCuP*

topology [197], i. e. a multidimensional "Cube" which captures content-similarity of the super-peers. The super-peers keep indexes for the content which is made available by the ordinary peers. Moreover, Edutella is aware of schema information which is shared among the super-peers and taken into account for semantic mapping and query optimization.

Structured Networks are highly scalable and employ a specific logical network topology, e. g. a ring, tree, or cube, with efficient access to the data that takes at most $\mathcal{O}(\log N)$ routing hops, where N is the number of peers in the network, for a data *lookup* operation, i. e. for finding the peer responsible for the requested data item. This is achieved by a specific data distribution, i. e. each peer is responsible for a certain portion of the data. Structured Peer-to-Peer networks commonly provide the data abstraction of a *distributed hash table* (DHT), i. e. key/value pairs are stored and retrieved with put and get functionality. Popular implementations are, for example, *CAN* [191], *Chord* [209], and *Pastry* [195]. Distributed hash tables have been studied extensively with respect to scalability, resilience, replication, churn, and optimization of network traffic for routing. Moreover, advanced querying support has been added, e. g. range and prefix queries in *P-Grid* [3, 5]. However, a major problem in DHTs is the efficient processing of complex (conjunctive) queries.

3.6.2 RDF in Structured Peer-to-Peer Networks

The scalability issues of unstructured Peer-to-Peer networks have led to more intense research in the area of storing RDF in distributed hash tables. This also shifts the focus from managing various RDF documents to storing individual RDF triples on the peers. Thus, the challenge for a distribution of RDF triples in a distributed hash table is to find a suitable data indexing scheme and allow for efficient query processing in terms of fast data access and minimal routing overhead. A good overview for RDF storage in structured Peer-to-Peer systems is given in [77].

Data Indexing

The key/value paradigm of distributed hash tables limits the flexibility when storing RDF triples. Moreover, the indexing strategy has to consider how query processing can be supported efficiently. With triple patterns as the basic building block of SPARQL queries, which can have different variations of bound variables, the common approach is to index each triple pattern three times, i. e. for subject, predicate, and object, and store it at the corresponding peers. To avoid a skewed data distribution among the peers, e. g. caused by popular predicates or types, some approaches, like GridVine [4, 58], use three more index combinations for $s-p$, $s-o$, and $p-o$. This also allows for better load distribution when executing queries but increases the size of the stored data.

Query Processing

Query Processing in distributed hash tables has to take different aspects into account: (1) the data is fully distributed in the network by hashing RDF triples, i. e. related information is not necessarily stored at the same peer, (2) the data is stored as key/value pairs and does not support complex data structures, and (3) although a single lookup takes $\mathcal{O}(logN)$ hops, routing many messages in a Web Scale environment can be extremely costly. Therefore, different query processing strategies are implemented for different query types, i. e. *atomic queries* (single triple pattern), *range queries* (triple pattern with filter expression on the object variable), *disjunctive queries*, and *conjunctive queries.* RDFPeers [42] is based on *multi-attribute addressable network* (MAAN) [43], with a Chord-like topology that uses locality preserving hashing of subject, predicate, and object. It allows for atomic queries, range queries with disjunctive object constraints on a single triple pattern, and conjunctive queries where all triple patterns have the same subject. GridVine [4, 58] is based on P-Grid [3] which supports range queries and prefix queries and semantic mapping between different schemas. The work of Liarou at al. [143] is similar to RDFPeers but adds conjunctive path queries by forwarding queries in a Chord ring and evaluating consecutive query patterns. Additionally, load balancing is achieved by hashing $s-p$, $s-o$, and $p-o$ as well as including variable bindings in forwarded queries. None of the approaches mentioned above is capable of evaluating complex SPARQL queries. *Page* [63] indexes triples with context (spoc) using the indexing scheme of YARS [99], i. e. SPOC, CP, OCS, POC, CSP, and OS indexes. The quad terms are hashed individually and combined with the index id. Queries use the same schema and have a query mask which indicates the positions of variables in a query. All peers with the same ID prefix (according to the mask) are asked for triples. The indexes lead to a clustering of the peer-id space. Page does not support conjunctive queries but just atomic triple patterns.

More recent implementations provide support for more advanced queries. Karnstedt et al. [127, 128] implemented an infrastructure for semantically-enriched data on top of distributed hash tables (UniStore[4]/P-Grid [129]) with schema mapping, SPARQL-like queries including prefix, range, edit distance, skyline. The prefix-search capability of P-Grid allows for maintaining less indexes, i. e. the $p-o$ index is used for a predicates only as well as for predicate-object combinations. The approach applies query shipping, query optimization of logical/physical plans and parallel execution of queries. It achieves quick response times through high parallelization and distribution across many nodes.

The approaches discussed above have not been shown yet to be applicable for really large RDF data. Only Kaoudi et al. [126] have evaluated their approach with a large dataset, i. e. LUBM [92]. RDF triples are stored three times and query order optimization based on selectivity estimation is applied. They implemented static and dynamic query optimization by processing queries in

[4] http://www.tu-ilmenau.de/dbis/research/the-unistore-project/

a chain (across peers) and avoiding cross products. A fully distributed mapping dictionary is used to replace terms with integers. Selectivity estimation is based on triple patterns and joins. Statistics are kept by each peer with histograms for term frequency and co-occurrence. The overhead for sending requests, mappings, and statistics has shown to be minimal compared with the number of messages required for query processing. The approach is implemented in the Atlas[5] [125] system.

3.7 Event-based Middleware

All scenarios described before were request-driven, i. e. the data is pulled on request from the location where it is stored. Event-based systems are completely different. Data providers publish their information as events and the data integration has to ensure that the data is processed on the fly, i. e. it has to be consumed in the moment the event is generated. A prominent example is sensor data or stock exchange. Due to the popularity of RDF there has also been some research effort in the direction of combining Semantic Web technologies with sensor applications [180, 20, 182] and publish it as Linked Data. There are different approaches for processing of RDF event data, i. e. RDF stream processing and publish/subscribe based event consumption.

3.7.1 RDF Stream Processing

Sensor networks produce continuous streams of data which have to be processed in real time. The combination with semantic Web technology allows to define domain-specific ontologies and model sensor events as structured data [44]. Walavalkar [224] speaks of *streaming knowledge bases* which pose new challenges for the data processing including reasoning. *Continuous SPARQL* (C-SPARQL) [19] is an extension of SPARQL with new features for evaluating long-standing queries on RDF streams, i. e. sequences of (RDF triple, time stamp) pairs. It introduces aggregation, window specifications (sliding or tumbling), and definitions for when a query should be computed. The result is a set of bindings or new RDF data streams. A different RDF stream processing approach is presented by Bolles et al. [35]. Their work is based on a changed semantics of SPARQL to integrate the evaluation of time stamps in every operator. However, this complicates the evaluation of queries and cannot be used in combination with common SPARQL query processing implementations. Moreover, their approach is also missing aggregation functions on the results.

[5] http://atlas.di.uoa.gr/

3.7.2 Publish/Subscribe Systems

Publish/Subscribe systems are designed for event-based producer/consumer scenarios, e. g. smart spaces, location-based services, or stock exchange. A general feature of these systems is the decoupling of producer and consumer, i. e. a consumer defines an interest in certain events with an event *subscription* and the producer publishes event *notifications* which are routed asynchronously through the event-based middleware to any interested consumer. A publish/-subscribe middleware is especially suitable for scenarios with many producers and many consumers.

Publish/Subscribe is suitable for Linked Open Data when there is an interest in notifications concerning new data being added or if data is updated, e. g. frequently changing data like product prices, continuous sensor data, or anything else that is suitable for stream-based data consumption. Subscription are typically defined as filters on topics or properties. Topics are organized in a tree hierarchy and they are assigned for the whole event notification. Topics are often too broad for a consumers interest. Property-based subscriptions are more specific as they select properties from the content of the event notification. Thus they are more flexible but the property-based matching is also more complex. Content based publish/subscribe systems also support more complex queries which is very similar to the evaluating conjunctive triple patterns in SPARQL queries. One such approach has recently been presented by Abdullah et al. [2]. It provides a translation of SPARQL queries to rules in *Rete* [80]. Rete is an efficient implementation of parallel pattern matching. All rules, i. e. query patterns, are arranged in a network tree. New facts, i. e. RDF triples are checked against the rules and stored in shared temporal memory if further rules have to be applied. The initial implementation shows that the Rete implementation can compete with static queries in Sesame and Jena. Event-based filtering is also known as *continuous queries* and can be realized on top of a peer-to-peer network [144].

3.8 Summary

This chapter presented different scenarios and approaches for Linked Data integration. First, several functional and non-functional requirements for RDF data integration were identified, which yield different infrastructure paradigms with certain advantages and disadvantages. Then, the most prominent infrastructure paradigms, i. e. data ware housing, search engines, federation, link traversal query processing, and peer-to-peer systems, (and event-based middleware), were discussed in detail.

Concerning the differences between materialized and virtual data integration it turns out that Linked Data integration requires high flexibility and scalability because of the large number of heterogeneous data sources. Hence, data maintenance in data warehouses and peer-to-peer systems is generally

too expensive to ensure up-to-date results. Another common constraint for most scenarios is the support for complex structured queries, e. g. through a SPARQL endpoint. In this sense, search engines and the link traversal query processing are too restricted and the latter approach may not be able to return complete results.

Consequently, federation is left as the remaining data integration paradigm which seems to fit all requirements best, especially with regard to virtual data integration and support of complex queries. Since Linked Data federation is similar to federated databases, there is a wide range of database technology from 30 years of research which can potentially be adapted. But there are also significant differences concerning the graph structure of RDF and the capabilities of distributed RDF data sources. Hence, there are still many open research questions. The next two chapters will specifically focus on data source selection and query optimization for distributed RDF data.

4

SPLENDID Data Source Selection

Federation infrastructures hide the complexity of data locality from the user and provide a query interface which can be used as if all data would be stored in one large database. A manual definition of sub queries with associated data sources, e. g. as provided by the SPARQL 1.1 federation extension [186] via the SERVICE keyword, is not the responsibility of the user. Instead, federated SPARQL queries should basically be defined like common queries on a local data source and the federation engine takes care of schema mappings and automatic distribution of sub queries to relevant data sources. Thus, the user does not need to know anything about the actual data distribution.

Automatic query splitting and data source mapping is challenging for queries on the Linked Data cloud because it is usually impossible to know in advance which data sources can contribute results for a query. Link traversal query processing [105] employs dynamic source discovery during query execution by resolving resource URIs in an iterative fashion starting with the URIs contained in a SPARQL query. However, this approach has certain limitations (cf. Sec. 3.5), especially with respect to query expressiveness and result completeness. The objective for Linked Data federation is to allow for executing complex SPARQL queries on an arbitrary number of Linked Data sources and return complete results. Therefore, it is necessary that each data source offers a common query interface, i. e. a SPARQL endpoint, and a data summary to give information about the stored data.

This chapter deals with the challenges of effective data source selection for federated SPARQL queries, e. g. suitable strategies for mapping triple patterns to data sources and specific aspects concerning the indexing of the required metadata. The novel SPLENDID approach [88] is presented which uses VOID descriptions for static data source selection. Finally, an evaluation on real Linked Data verifies its effectiveness and further optimizations are discussed.

4.1 Federated Linked Data Management

The Linked Data cloud contains many different datasets and it is constantly growing as new data sources are added. Although it is almost impossible to know every dataset, a federation system needs to learn about “relevant” datasets and their content in order to decide where queries have to be sent to. Hence, a data source manager component takes care of the integration and maintenance of different data sources and their metadata. SPLENDID relies completely on SPARQL endpoints and does not consider URI resolution as used in *link traversal query processing* (cf. Sec. 3.5). The latter is limited with respect to the query expressiveness and, due to the focus on pure Linked Data principles, it requires essentially a different query processing implementation which employs many URI lookups. Instead, SPARQL endpoints allow for using complex queries on a larger dataset. Thus, the source selection has to rely on a data source catalog which contains information about all known data sources. A data source entry includes the URL of the SPARQL endpoint and additional metadata, e. g. size of the dataset, used vocabularies, and schema information which is used for mapping query expressions to data sources. The data source integration can be done statically or dynamically. Either way, the responsibility of a data source manager is to retrieve all data source descriptions and integrate them in the local data source catalog.

4.1.1 Static and Dynamic Data Source Integration

There exist basically two different approaches for integrating data sources in a federation, i. e. with static definitions or dynamic discovery. Both approaches have advantages and disadvantages but it typically depends on the federation scenario, which of them is favorable. In addition, it is also possible to use a hybrid combination of static and dynamic data source integration.

Static Source Definitions A static definition of data sources is applicable for specific domains when all relevant datasets are already known at setup time, e. g. an expert in the life sciences domain may need to integrate particular scientific data like UniProt [18] or Drugbank. The setup is typically done manually by listing all relevant data sources with the respective SPARQL endpoints and a summarization of their content.

Dynamic Source Discovery Due to the large size of the Linked Data cloud it is essentially impossible to know all relevant data sources in advance (unless the federation is restricted to a particular subset). Hence, a dynamic discovery approach offers the opportunity to find new data sources on the fly. However, a query federation system cannot employ URI lookups during query execution but needs to find SPARQL endpoints of data sources which are capable of returning results for certain query patterns. Currently, only the CKAN repository and search engines like Sindice [178] can be used to gather such information. But it is not guaranteed that

the returned information is useful or complete, e. g. that links to SPARQL endpoints are available. Furthermore, data summaries are often missing.

SPLENDID uses the static approach because it requires certain statistics which are best generated in a pre-processing step. Collecting such statistics on the fly is currently not possible in an efficient way. Changes of data sources are in general problematic because it requires sophisticated update strategies to keep a data source catalog up to data.

4.1.2 Describing Data Sources with VOID

In order to determine if certain datasets will be able to produce results for a query expression the source selection needs information about the content of all federated data sources. The creation of individual hand-crafted data source descriptions, like in DARQ [188], is infeasible for the Linked Data cloud. Moreover, extensive pre-processing steps, as employed for Q-Trees [101] or graph structure indexes [211, 215, 150, 23] can be expensive as well. Ideally, the data sources descriptions should be available in a compact representation from the publishers of the datasets. However, they also have to consider cost and benefit of providing such information.

Although the source selection benefits most from detailed data source descriptions, their size should be limited for different reasons. First, an efficient data source selection needs to keep the descriptions of all federated data sources in memory. Moreover, the generation of the data source descriptions should be straightforward and require only little extra effort for the data provider. Otherwise, it will outweigh the benefits, especially in case of changing datasets. For example, maintaining information about all resources in the Linked Data cloud is impossible. Hence, data summaries are typically focused on schema-level information which is easy to generate and relatively stable with respect to data changes. Finally, the source description should be based on standard semantic web technology, preferably RDF, and allow for integration of statistical information for query optimization and be extensible for future requirements. Theses requirements can currently only be satisfied with the *Vocabulary for Interlinked Datasets* (VOID) [9]. It provides means for describing Linked Data sources with general metadata and supports basic statistical information, e. g. frequency counts.

Figure 4.1 shows an example VOID description for the ChEBI dataset. It starts with general dataset information like title, description, homepage, and URL of the SPARQL endpoint. Additionally, it can include references to the used vocabularies or notes about the creator. The second section deals with basic statistics, like the triple count, the number of distinct properties, entities, subjects, or objects. More detailed statistics are given with property partitions and class partitions. They define the number of occurrences of predicates and types in the RDF triples of the data sources. All statistical data can be created automatically.

```
 1  :ChEBI a void:Dataset ;
 2
 3  # general information
 4  dcterms:title "ChEBI" ;
 5  dcterms:description "Chemical Entities" ;
 6  foaf:homepage
 7      <http://chebi.bio2rdf.org/>
 8  void:sparqlEndpoint
 9      <http://chebi.bio2rdf.org/sparql> ;
10
11  # simple data statistics:
12  void:triples "7325744" ;
13  void:entities "50477" ;
14  void:properties "28" ;
15  void:distinctSubjects "50477" ;
16  void:distinctObjects "772138" ;
```

```
17  # entity count per concept
18  void:classPartition [
19      void:class chebi:Compound ;
20      void:entities "50477" .
21  ] ;
22
23  # triple count per predicate
24  void:propertyPartition [
25      void:property bio:formula ;
26      void:triples "39555" ;
27  ] , [
28      void:property bio:image ;
29      void:triples "34055" ;
30  ] , [
31      ...
32  ] .
```

Fig. 4.1. VOID description excerpt for the ChEBI dataset containing general information and statistical data, like total triple count and number of occurrences of predicates and instances. (namespace are omitted for better readability, see http://void.rkbexplorer.com/ for more examples)

4.1.3 VOID Generation

VOID descriptions are not widely available yet because there are only a limited number of application scenarios and most data providers do not see a benefit yet. A federation infrastructure based on VOID descriptions may provide the right incentives for a broader support in the future. However, for now, VOID descriptions usually have to be obtained in a pre-processing step. The generation of statistical information for VOID descriptions is straightforward. It basically requires counting number of triples, properties, distinct subjects, and so on. There are two typical scenarios for creating the statistics, i. e. by analyzing data dumps or by sending specific queries to the SPARQL endpoint.

Data Dump Analysis This is the most efficient way for generating VOID descriptions. Schema-related item counts and statistics can basically be extracted with a single pass over all RDF triples. More sophisticated structure analysis and the processing of large datasets may require specialized extraction algorithms [34, 131].

Query-based Analysis Data dumps may not be available for all data sources. Hence, the required information has to be extracted in a different way, i. e. through specifically crafted queries which are sent to the SPARQL endpoints. Such queries are typically aggregate queries which ask for all predicates and types with their respective number of occurrences in the dataset. Thus, a SPARQL endpoint has to support SPARQL version 1.1 [98]. Otherwise, item counts cannot be extracted in an efficient way.

4.2 Data Source Indexing

All schema-related information and statistical data from the VOID descriptions are stored in the data catalog in order to make them easily accessible for the source selection and query optimization. Hence, the data catalog has to support indexing and retrieval of several data types and counts. For the data source selection it keeps a mapping relation $M \subseteq \{(iri, ds) \,|\, \forall\, iri \in I \wedge ds \in D\}$ from schema-specific RDF IRIs (cf. Def. 2.4) to matching data sources. Although inverted lists are commonly used for such data mappings SPLENDID employs RDF for storing all information in the data catalog. Thus, the VOID descriptions can be directly imported and lookups for data sources can be expressed as SPARQL queries. However, maintaining a data catalog for many Linked Data sources can be challenging due to the large amount of data. Typically, the data catalog should fit into memory. But depending on the kept information it may be necessary to store specific data indexes on disk or use compression techniques in order to reduce the overall size. Following is an overview of common constraints for the data catalog.

Index Size The size of a data source catalog influences the performance of the source selection, e. g. expensive I/O operations can be avoided if all mappings are kept in memory. The schema information found in VOID descriptions is relatively small in size and can be easily maintained in memory. More detailed information typically has to be summarized or compressed. In addition, the cost for operations on the data structures will increase with the index size.

Level of Detail A precise data source selection only includes mappings to datasets which can actually return results for the respective query expressions. But the memory size limitations make it essentially impossible to maintain very detailed information, e. g. mappings for all URIs in every Linked Data source. Hence, there is a trade-off between the level of detail and the mapping precision. VOID descriptions are focused on schema-level data. This offer a good recall for schema-based queries but query expressions with instance-level restrictions will have a lower mapping precision.

Maintenance Cost Data sources in the Linked Data cloud can change more or less frequently, i. e. resources are updated or new data will be added. To ensure high precision and recall for the data source mappings the indexed data in the catalog has to be updated as well. This can be a complicated task depending on the number of different data sources, the amount of indexed data, and the level of detail, e. g. it may require to repeat expensive operations like a data dump analysis. VOID description are rather easy to maintain, since the updated information can simply replace the old indexed data.

With respect to index size and maintenance cost VOID statistics are a good choice for building a data catalog. Although the SPLENDID source selection can provide an effective data source mapping based on VOID descriptions only,

it would also be possible to incorporate and combine further data source information in a data source catalog. Following section gives a short overview of state-of-the-art indexing approaches for (distributed) RDF datasets and summarizes specific advantages and disadvantages regarding their applicability for the data source selection. As it turns out, it is generally not possible to find an index combination which is optimal in every aspect.

4.2.1 Schema-level Index

Properties and Classes

A schema index basically stores information about the use of predicates and class types found in the data sources. Therefore, two inverted indexes I_p and I_t are maintained. The index size depends on the number of predicates, types, and data sources. The data distribution is typically skewed, because popular predicates, like rdf:type and rdfs:label occur in almost every data source while other predicates or types may just be used in one dataset. Predicates and type definitions can be extracted directly from a dataset's RDF triples. However, some data sources may also provide an explicit schema definition in a separate ontology, e. g. including information about inheritance of classes and properties, like rdfs:subClassOf or rdfs:subPropertyOf, and restrictions to domain and range of relations. Such additional schema information may be indexed as well and used for schema mapping. SPLENDID [88] and DARQ [188] are two examples for federated query optimization approaches which rely on schema indexes.

Namespace Index

VOID descriptions allow to include information about the vocabularies used by a dataset. Such information may be helpful for the source selection if no other metadata is available [8]. Namespace indexes contain the URI namespaces which are exployed for defining the concepts of a vocabulary. However, vocabularies are primarily used for schema definitions and specific vocabularies, e. g. rdf, rdfs, foaf, dublin core, are more popular than others and used by many datasets. Moreover, a dataset typically does not include all concepts of a vocabulary. Hence, the data source mapping is typically coarse and in combination with a schema index it does not provide any extra information. An exception are vocabularies which are used for instance-level data in a dataset.

Currently, VOID descriptions with vocabulary information are rarely available. One possible source is the CKAN repository[1] , but the provided list of vocabularies can be incomplete. Hence, an analysis of RDF data dumps would usually be necessary in order to obtain all vocabulary data. But then it is also possible to extract all schema-related data for a schema index. Instance-specific namespaces cannot be described separately in VOID descriptions.

[1] http://ckan.org/

4.2.2 Instance-level Index

Resource Index

The overall number of resources (URIs) in a set of Linked Data sources is typically too large for in-memory data catalogs. However, since SPARQL queries may contain specific resource URIs in the triple patterns it would be useful to have such information available for the source selection. There exist different approaches which apply compression techniques, but most of them are focused on maintaining statistical data for query optimization [14, 171]. For the purpose of source selection, only the Q-Tree-based data summaries [101] and the PARTTree extension [185] are applicable. A Q-Tree is a three dimensional data structure which uses hashing of (s, p, o) triples in combination with histogram-like bounding boxes. Each bucket contains a list of associated data sources and frequency counts. The Q-Tree can store information for an arbitrarily large number of Linked Data sources. But the cost for generating such data summaries is high and typically requires off-line process of data dumps, which may not always be available.

Literal Index

All aforementioned indexes store mappings for URIs. But literals may also occur in SPARQL queries and triple patterns with a literal are typically very selective because in most cases there is just one data source which contains the literal. However, as with the resource index, it is usually infeasible to store a mapping for each literal and its respective data sources. Moreover, literals can contain comprehensive descriptions, even complete text documents, and they can be annotated with language tags. Therefore, a full-text indexing, e. g. as provided by Apache Lucene[2], would fit better. But index size and maintenance cost will be increased. In addition, it is not possible to extract such information from a data source without pre-processing of a data dump.

A special case are literals with data types, e. g. integer, float, boolean, or date. In combination with a specific predicate, they describe certain attributes of a resource, like a person's age or a city's population. Such attributes are often matched by filter expression in SPARQL queries. Hence, the source selection could basically benefit from additional statistical information, i. e. by excluding data sources which cannot provide results for the defined attribute ranges. Traditional databases commonly use histograms [120] to capture statistical information about such attributes. But employing histograms for federated RDF data sources is more complicated. Currently, there exist no suitable solution for representing RDF data statistics in a concise and useful way, e. g. Stocker [208] and Langegger [139] apply histograms on RDF data but they do not differentiate between data types, thus, URIs, literals, and numbers are merged. Other options for comprehensive representation of item sets, besides

[2] http://lucene.apache.org/

histograms, are *Bloom filters* [33, 38], *compressed Bloom filters* [160], or *hash sketches* [78].

4.2.3 Graph-level Index

Triple Pattern Index

The aforementioned indexes cover individual RDF terms which occur as constants in triple patterns. Hence, common source selection approaches [188, 88] for term combinations, e. g. (?x owl:sameAs dbpedia:Berlin), compute the intersection of the sets of selected data sources. However, the result can include false positives since the selectivity of the combination of two RDF terms is typically higher than the individual selectivities. Hence, centralized triple stores, like RDF3X [172] or Hexastore [226], index all permutations of RDF term combinations and apply sophisticated compressions techniques to reduce the index size. But the computation is prohibitively expensive for a federation system. The *Q-Tree* [101] is an alternative data structure (cf. instance index). However, since VOID does not support the definition of RDF term combinations such data cannot be easily exchanged between data sources.

Structure Indexes

Joins in a SPARQL query define restrictions on the result set, i. e. a partial result for one triple pattern may not be included in the final result set if joined with other data from the same or another data source. Detecting such situations and removing the respective data sources in the source selection will simplify the query optimization and speed up the query execution. However, additional information is needed in order to find out which data sources can actually return results for a join combination of triple patterns. As SPARQL queries can have arbitrary graph patterns a structure index has to store RDF sub graph descriptions with a varying number of joined triple patterns. Following categorization describes a number of related approaches.

Path Index Stuckenschmidt et al. [211] employ a path index to store schema-level information about connected RDF triples. Each path is defined by the bound predicates and contains a set of matching data sources and the number of occurrences. However, SPARQL queries also contain star-join patterns which cannot be captured by such an index.

Resource Index *Characteristic Sets* [171] cover resource-centric star-join patterns. While initially designed to maintain schema-level statistics for query optimization they can be easily extended with information about respective data sources. Yet, there is no implementation for federated data source selection. A slightly different approach [79] uses equivalence classes based on the similarity of schema-level data.

Graph Index A catalog for frequent graph patterns is useful for mapping data sources to arbitrary SPARQL query patterns. In fact, there exist several graph mining approaches for identifying frequent sub-graph patterns in RDF data [215, 150, 23]. But the in-depth analysis of the datasets is computationally expensive.

Structure-based indexes are commonly generated for individual dataset. It would be beneficial for the federation of Linked Data sources if such information can also be collected for graphs across data sources. But building and maintaining such an index for joins of any number of triple patterns in any possible combinations across any data source is prohibitively expensive.

4.3 Data Source Selection

The performance of federated query processing on Linked Data sources is significantly influenced by the network communication overhead. Hence, the number of requests which have to be sent to remote data sources should be minimal and involve only data sources which can actually return results. Datasets which cannot contribute to the final result set, i. e. false positives, will increase the number of request without adding new results, and thus, decrease the query processing performance. However, a-priori it is unclear which datasets can contribute results for which part of a query. Each triple pattern in a SPARQL query may potentially be answered by different data sources (cf. Fig. 4.2). But without executing queries for all triple patterns on all data sources it is essentially impossible to determine the exact set of relevant datasets. Therefore, the SPLENDID source selection algorithm has to rely on a data source catalog which contains knowledge about the data sources. Moreover, it employs optimization heuristics in order to obtain complete results with a minimal number of sub queries.

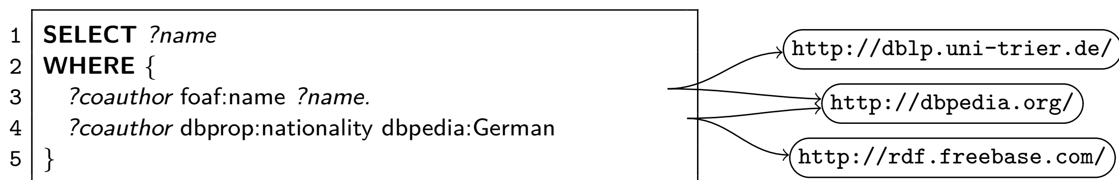


Fig. 4.2. A SPARQL query with two triple patterns which are mapped to three different data sources.

SPLENDID uses a static data source integration approach. Hence, all required VOID description are collected and indexed in a pre-processing step. The data source catalog fits into memory and allows for fast computation of mappings from triple patterns to data sources. Yet, it is extensible with additional (statistical) information which can be used for the query optimization. The optimal trade-off between query performance and result completeness

largely depends on the setup of the federation system and the metadata that is available during the data sources selection. SPLENDID has the primary focus on result completeness because an exact estimation of the number of results for the different data sources is complicated based on the schema-level VOID descriptions. Following, is a detailed explanation of the source selection strategy which is divided into three individual phases. First, schema-based data source mappings are determined for each triple pattern. Thereafter, a refinement of the data source mappings is done with the help of SPARQL ASK queries. Finally, the overall number of request is minimized by building sub queries containing multiple triple patterns with common data sources.

4.3.1 Schema-based Source Selection

SPLENDID employs data source selection per triple pattern using the schema information from the VOID descriptions. First, all triple patterns with a bound predicate URI and those with rdf:type as predicate and a bound class URI as object will be processed. A lookup of a predicate or type URI in the predicate or type index, respectively, will return the associated set of data sources. If a triple pattern does not have any other bound variables than predicate or rdf:type and class the returned lookup result represents the exact data source mapping. Otherwise, only a subset (potentially none) of the data sources may be able to return results because of the additional restriction applied by another bound variable. The obtained data source mappings are assigned to the respective triple patterns, which are separated into the set of triple patterns with exact mappings and the ones without exact mapping. Finally, all remaining triple patterns, i. e. without a bound predicate, will be mapped to all known data sources because it is not possible to determine any restriction based on the schema-level index. This set of triple patterns should generally be small since common SPARQL queries typically contain a bound predicate [82, 184]. However, requests to all data sources are highly inefficient. Hence, these triple patterns will also be added to the set of triple patterns without exact mappings for further processing in the following refinement phase.

4.3.2 Data Source Pruning with ASK Queries

Resource URIs and literals are not covered in the VOID-based indexes. Therefore, a triple pattern with a mix of schema-specific and instance-specific terms, such as (?x rdfs:label "Berlin"), can only be mapped to the data sources which are associated with the schema-related RDF term, i. e. rdfs:label. While rdfs:label is a popular predicate and used in many datasets, the literal "Berlin" is rather selective, i. e. it occurs only in a few datasets and not necessarily in combination with rdfs:label. The selection of false positives is problematic because of higher communication cost and longer query execution time without getting additional results. Moreover, the query optimization will become more

complex and finding the optimal query execution plan may not even be possible. Therefore, the set of selected data sources needs to be pruned for triple patterns which contain bound RDF terms beyond schema information.

SPLENDID employs SPARQL `ASK` queries to validate each triple pattern in the list of non-exact mappings. A SPARQL `ASK` query contains the same query expression (in the `WHERE` clause) as a SPARQL `SELECT` query but the result is either *true* or *false* depending on the ability of the data source to return results for the query expression. Consequently, each data source which returns *false* will be removed from the set of data sources for the respective triple pattern. In the worst case this approach can potentially double the number of sent requests, i. e. a SPARQL `ASK` query and SPARQL `SELECT` query for each combination of triple pattern with selected data source. However, in reality there will be a certain number of data sources which can be pruned with initial `ASK` queries. Moreover, the cost for sending an `ASK` query is much lower than the cost for the actual `SELECT` query, especially if intermediate result data needs to be transferred. Therefore, the use of ASK queries can significantly speed up the whole query execution by avoiding expensive requests with a few cheap ones. Algorithm 1 shows in detail how the source selection is done.

Algorithm 1 Source Selection for triple patterns using VOID and `ASK` queries.

Require: I_p; I_τ; $D = \{d_1, \ldots, d_m\}$; $T = \{t_1, \ldots, t_n\}$ // indexes, data sources, and triple patterns
1: **for** each $t_i \in T$ **do**
2: $sources = \emptyset$
3: $s = subj(t_i)$; $p = pred(t_i)$; $o = obj(t_i)$
4: **if** ! $bound(p)$ **then**
5: $sources = D$ // assign all sources for unbound predicate
6: **else**
7: **if** $p =$ rdf:type $\wedge$ $bound(o)$ **then**
8: $sources = I_\tau(o)$
9: **else**
10: $sources = I_p(p)$
11: **end if**
12: **end if**
13: // prune selected sources with `ASK` queries
14: **if** ! $bound(p) \vee bound(s) \vee bound(o)$ **then**
15: **for** each $d_i \in sources$ **do**
16: **if** $ASK(d_i, t_i) \neq true$ **then**
17: $sources = sources \setminus \{d_i\}$
18: **end if**
19: **end for**
20: **end if**
21: **end for**

4.3.3 Sub-Query Optimization

The result of the source selection is a list of mappings where each triple pattern has a set of selected data sources (i. e. SPARQL endpoints). A naïve federation approach could basically send each triple pattern individually to the assigned data sources and then aggregate the returned results. However, an important aspect of the query optimization is to minimize the number of requests which have to be sent. For example, if multiple triple patterns have been assigned to the same set of data sources it might be possible to aggregate them into a single sub query. But certain constraints have to be considered. There are situations where the combination of triple patterns can lead to incomplete results. Moreover, cross products may be introduced which can yield large intermediate result sets such that large amount of data have to be transferred over the network.

Figure 4.2 illustrates a SPARQL query with two triple patterns and three selected data sources, namely http://dblp.uni-trier.de/, http://dbpedia.org/, and http://rdf.freebase.com/. At least three sub queries are required, i. e. one for each data source. However, the combination of both triple pattern in a sub-query for http://dbpedia.org/ can yield an incomplete result set, e. g. if DBpedia contains results for the first triple pattern which can be joined with results of Freebase for the second triple pattern. Hence, triple patterns with multiple assigned data sources usually cannot be combined but have to be sent individually, thus, ensuring that all intermediate results can be joined across data sources. The SPLENDID source selection relies on three different heuristics to group triple patterns while trying to ensure result completeness as much as possible.

Exclusive Groups. If a single data source is exclusively selected for a set of triple patterns all of them can be combined into one sub query. Such a set of triple patterns has been referred to as *exclusive group* in FedX [203]. The join structure of the triple patterns in exclusive groups has no effect on the result completeness. However, for performance reasons cross products should be avoided. Hence, the exclusive groups need to make sure that the joined triple patterns form a connected graph, i. e. each triple pattern contains at least one variable which is shared with another triple pattern in the exclusive group.

Resource Groups. Queries for resources typically have a star-shaped graph pattern where the respective variable used for matching the resources occurs in the subject position of the respective triple patterns. Due to the fact that resources are identified with a unique URI, which is defined by the data provider, all of the information for a specific resource can usually be found in one data source. Moreover, it is common practice to use owl:sameAs links and a distinguishable URIs when similar resources are defined in different namespaces. Hence, it makes sense to combine such triple patterns, especially if they contain popular predicates, like `rdfs:label` or

foaf:knows, which are matched by many data sources. In this case, the intersection of the data sources of the relevant triple patterns is computed. This heuristic actually sacrifices result completeness if information about the same resource is defined in different data sources.

SameAs Groups. SameAs links are the most common type of links between datasets in the Linked Data cloud. Thus, the source selection for triple patterns with the predicate owl:sameAs typically returns a large set of matched data sources. Assuming that all owl:sameAs links are defined in the same dataset as a link's source entity, it is possible to group all triple patterns, which contain the variable used for matching the respective entity, and reduce the data source sets by calculating the intersection. This approach has to be used with caution because in the presence of 3rd party datasets with external owl:sameAs definitions it is not possible to ensure result completeness. Moreover, besides owl:sameAs, other predicates may be used as well to define links between datasets. In such cases a combination of triple patterns like { ?x foaf:knows ?y . ?y owl:sameAs ?z } should not be restricted to the same data sources.

The SPLENDID source selection employs all aforementioned heuristics in order to minimize the number of sub queries which have to be sent to the SPARQL endpoints. Its effectiveness depends on the characteristics of the given queries and the number of involved data sources. Therefore, following evaluation investigates the effect of the source selection for a number of representative SPARQL queries.

4.4 Evaluation

The results of the source selection have a large influence on the distributed query execution performance. Hence, different settings of the SPLENDID source selection strategy are compared in an evaluation in order to analyze the effects of the heuristics and optimization steps and to show that suitable data source mappings based on VOID descriptions can be produced. However, the evaluation cannot just measure the query execution and response times because they depend too much on an effective query optimization and efficient query execution. Instead, the evaluation measures the source selection quality based on the number of selected data sources and the number of generated sub-queries, i. e. requests which have to be sent over the network.

Three variations of the source selection are compared. The baseline employs only the information about predicates from the VOID descriptions to map triple patterns to data sources. Sub queries are created using exclusive groups if possible. The second configuration also takes type information from the VOID descriptions into account. Finally, the third one combines triple patterns with owl:sameAs in order to reduce the overall number of sub queries.

Table 4.1. FedBench evaluation dataset statistics

Data Set	triples	links	predicates	types
DBpedia 3.5 subset	43.6 M	61.5 k	1063	248
GeoNames	108.0 M	118.0 k	26	1
LinkedMDB	6.2 M	63.1 k	222	53
Jamendo	1.1 M	1.7 k	26	11
New York Times	0.3 M	31.7 k	36	2
SW Dog Food	0.1 M	1.6 k	118	103
KEGG	1.1 M	30.0 k	21	4
ChEBI	7.3 M	–	28	1
Drugbank	0.8 M	9.5 k	119	8

4.4.1 Setup

The FedBench [198] benchmark suite is used to evaluate the source selection for distributed queries on real Linked Data sources. It provides snapshots of 10 linked datasets, which are carefully chosen with respects to aspects like dataset size, data diversity, and the number of interlinks. An evaluation on the original SPARQL endpoints not possible as it does not allow for an objective comparison due to unpredictable reliability and varying latency.

The benchmark queries resemble typical requests on the datasets and their structure ranges from simple star and chain queries to complex graph patterns. All queries cover at least two different data sources. Table 4.1 gives details about the size of the data sets along with statistical information. The number of sources which contribute results to a query and the number of result tuples are shown in Tab 4.2.

Table 4.2. Number of data sources which are covered in FedBench's cross domain (CD) and life science (LS) queries and the number of results per query.

	CD1	CD2	CD3	CD4	CD5	CD6	CD7	LS1	LS2	LS3	LS4	LS5	LS6	LS7
sources	2	2	5	5	5	4	5	2	4	2	2	3	3	3
results	90	1	2	1	2	11	1	1159	333	9054	3	393	28	144

The dataset snapshots were distributed across five 64bit Intel® Xeon® CPU 3.60GHz server instances running a Sesame 2.4.2 triple store implementation with each instance hosting the SPARQL endpoint for one life science and for one cross domain dataset. All 14 queries for the life science and cross domain datasets of FedBench were executed on these SPARQL endpoints. The query processing was done on a separate server instance with 64bit Intel® Xeon® CPU 3.60GHz and a 100Mbit network connection.

4.4.2 Results

The compared source selection strategies are based on 1) VOID predicates and exclusive groups, 2) VOID predicates + type and exclusive groups, and 3) VOID predicates + type and exclusive groups + owl:sameAs groups. For all life science and cross domain queries the number of selected data sources and the number of requests to SPARQL endpoints, which are necessary to execute the generated sub queries, are measured. Figure 4.3 shows the results. Lower values mean less selected data sources and fewer requests, and thus, faster query execution.

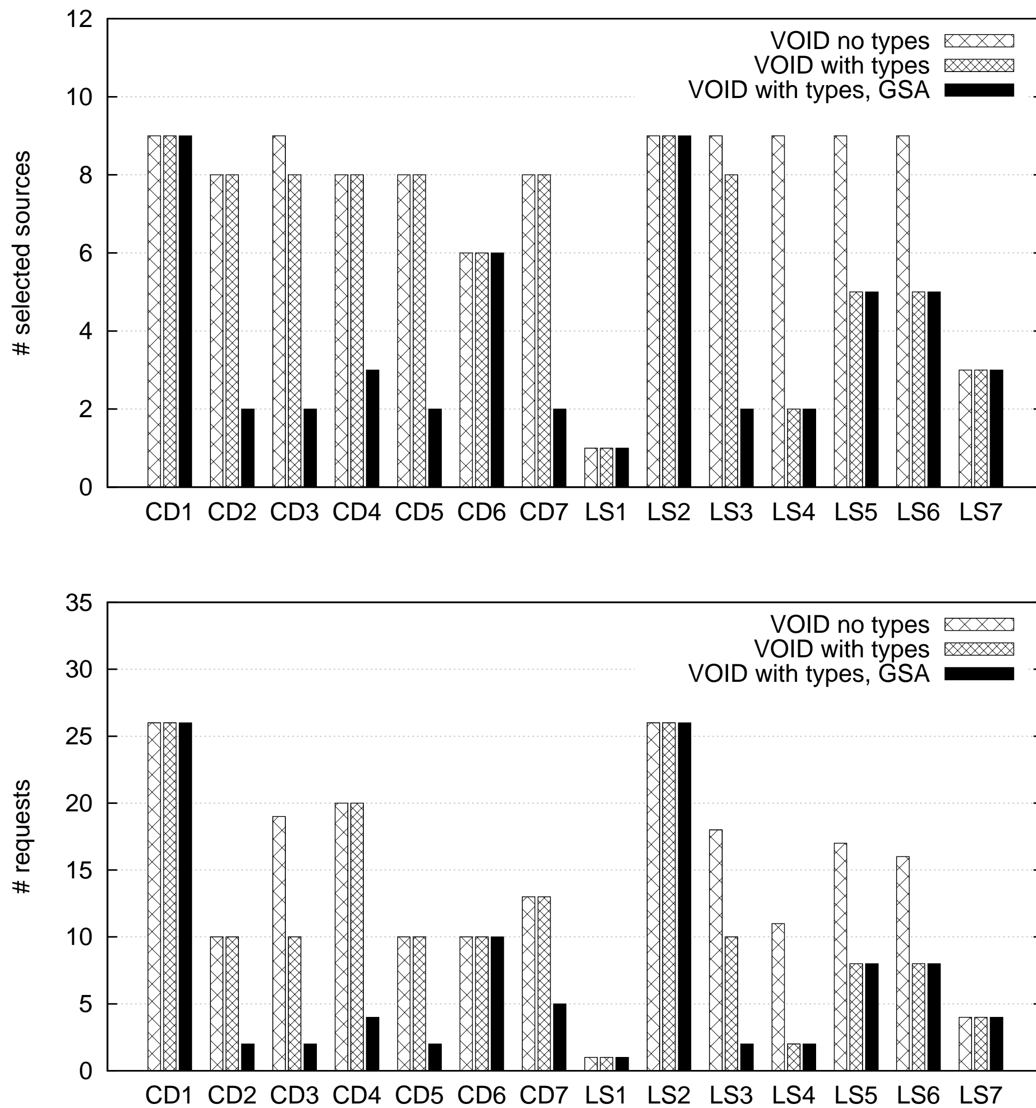


Fig. 4.3. Number of selected data sources (above) and number of SPARQL endpoints requests (below) when using VOID statistics with or without type information, grouping owl:sameAs patterns (GSA)

Life science and cross domain queries show different results for the compared source selection strategies. Regarding the number of selected data sources, Figure 4.3 shows that type information has no effect for the cross domain queries (except for CD 3) because they do not contain any triple pattern with rdf:type. For the life science queries, one to seven data sources can be excluded if type information is used. In contrast, the grouping of triple patterns with owl:sameAs has no effect for life science queries (except for LS3), as such triple patterns are not part of the queries. In contrast, for the cross domain queries one to five data source can be ignored. Queries CD1 and LS2 are special because they contain a triple pattern with no bound variable. Hence, all sources have to be selected in any case in order to obtain complete results.

Figure 4.3 also shows the number of requests which have to be send to execute all sub queries. They depend on the number of joins in a query. On the other hand the aggregation of triple patterns in exclusive groups can reduce the number of sub queries. But this depends on the selected number of different data sources, i. e. for less selected data source it is more likely that multiple triple patterns are part of an exclusive group.

The presented results show that type information is generally important for the source selection. SameAs links between datasets can also be exploited to reduce the number of selected datasets. But the result completeness may be sacrificed if the link definition is not located in the same dataset as the link's subject. The FedBench datasets contain no third-party links. Therefore, the SameAs grouping can be safely applied in this case.

In general, the effects of the optimization depend a lot on the query characteristics. There are significant differences for the life science and cross domain queries. In the case of FedBench one query set has many rdf:type triple patterns the other includes owl:sameAs. Thus the optimization can improve only one of the two cases. The results are clearly visible in the charts.

4.5 Summary

The SPLENDID source selection approach follows two primary objectives, 1) maximization of result completeness and 2) minimization of the number of sub queries such that an efficient query execution can be done afterwards. The first goal requires to identify all datasets which can provide results for a query, the second considers the network communication cost for multiple requests because it has the largest impact on the query execution time. Therefore, the main challenges are an optimal utilization of compact data source descriptions in order to determine mappings from query expressions to data sources and for building sub queries.

The SPLENDID source selection employs a mapping of triple patterns to data sources using VOID based schema information alone, does a refinement with SPARQL ASK queries, and applies sub query optimization by creating

suitable triple pattern groups. More detailed source descriptions can theoretically help to improve the quality of the source selection result but the maintenance overhead for managing such source description can be very costly for the general Linked Data scenario. In order to show that the aforementioned requirements are met an evaluation on real datasets from the Linked Data cloud with representative SPARQL queries was conducted. A variation of source selection strategies was compared to assess the differences with respect to the number of selected data sources and the number of requests which are necessary to evaluate all sub queries. The results led to the conclusion that VOID based source selection is a viable solution for Linked Data federation. However, the overall effectiveness depends on the actual query characteristics.

5

SPLENDID Cost-based Distributed Query Optimization

Research on query optimization has a long history in the database community. The main objective is to determine the best query execution plan with respect to specific optimization criteria, like query execution time or result completeness. Information retrieval in the Semantic Web basically deals with the same objectives when executing complex SPARQL queries on RDF data. Therefore, the adaptation of suitable query optimization strategies known from centralized and distributed relational databases [132] seems to be reasonable. However, due to differences between the relational data model and RDF graphs as well as certain limitations of SPARQL endpoints, mainly centralized RDF triple stores have been implementing common database techniques for query optimization so far. Only recently, due to the growing number of freely available Linked Data sources and limitations of centralized data warehouse approaches (cf. Sec. 3.2), there has been an increasing interest in distributed SPARQL query processing. Especially federation-based infrastructures for Linked Data have received more attention lately.

Compared to traditional distributed/federated databases, the federation of interlinked RDF data sources has to deal with new challenges, e. g. highly autonomous and heterogeneous data sources from various domains containing graph data with different levels of structuredness[1]. Therefore, an adaption of common database approaches for query processing across federated Linked Data (via SPARQL endpoints) is not straight forward. In this chapter the architecture, optimization strategy, and implementation of SPLENDID [88] will be presented. It is a distributed query optimization approach for Linked Data

[1] The structuredness of RDF datasets ranges from highly structured data (like in relational databases) to hardly structured data, i. e. datasets without explicit schema and different resource types with varying properties

federation which employs concepts of federated databases [206] in order to allow for efficient query optimizations of distributed SPARQL queries. It follows the *optimize-then-execute* paradigm and utilizes *Dynamic Programming* [205], which is a well-known query optimization strategy in relational databases.

This chapter starts with an overview of common concepts and strategies of (distributed) query optimization approaches in relational databases. Section 5.2 highlights the major challenges for distributed query optimization in general and specifically for federated Linked Data. Then, in Sec. 5.3 the SPLENDID approach to federated SPARQL query optimization is presented. Details of the cost-based join-order optimization for SPARQL basic graph patterns are elaborated in Sec. 5.4. Section 5.5 presents evaluation results and compares SPLENDID with other state-of-the-art RDF federation implementations. A summarization and discussion of future challenges is given in Sec. 5.6.

5.1 Query Optimization in Traditional Databases

Query optimization for Linked Data faces similar challenges as in distributed and federated databases [132, 206]. But there are also significant differences which hinder a direct adoption of successful database technology. In order to better understand these differences it is necessary to first get an overview about important concepts and query optimization techniques which have been developed in over 30 years of database research.

5.1.1 Query Plans

A query plan is the logical representation of an algebraic query and defines the order in which the query operators should be executed. It is commonly represented as a tree. Each tree node is either a binary operator (e. g. join) or an unary operator (e. g. selection, projection, filter expression). Nodes take the output of their child nodes as input. The result is passed to the respective parent node. The root node produces the final result set. Leaf nodes define so called *access paths*, which specify how the input data is obtained from the data sources. The notion of *input relations* originates from database tables and refers to the set of tuples which are used as input for a query operator.

There are two different approaches for scheduling the execution of query operators, the *iterator-based* model and the *data-driven* model [65]. In the *iterator-based* evaluation each query operator takes control over its child nodes. In a top-down fashion, starting at the root node, each query operator signals start and end of query precessing to its child nodes and explicitly fetches each tuple from the input relations. Thus, the performance is dominated by the query operators with the lowest data rate. The *event-driven* evaluation works in a bottom-up fashion propagating result tuples to the parent node when they become available and using queues to organize the input data for each operator.

Query Plan Structure

Query plans can have different tree structures. Figure 5.1 shows an example query with two different query plan representations, namely *left-deep tree* and *bushy tree*. The first corresponds to the exact transformation of the sequential order of the query patterns while the latter defines an alternative operator order with different sub trees. Both types of query plans have certain advantages and disadvantages.

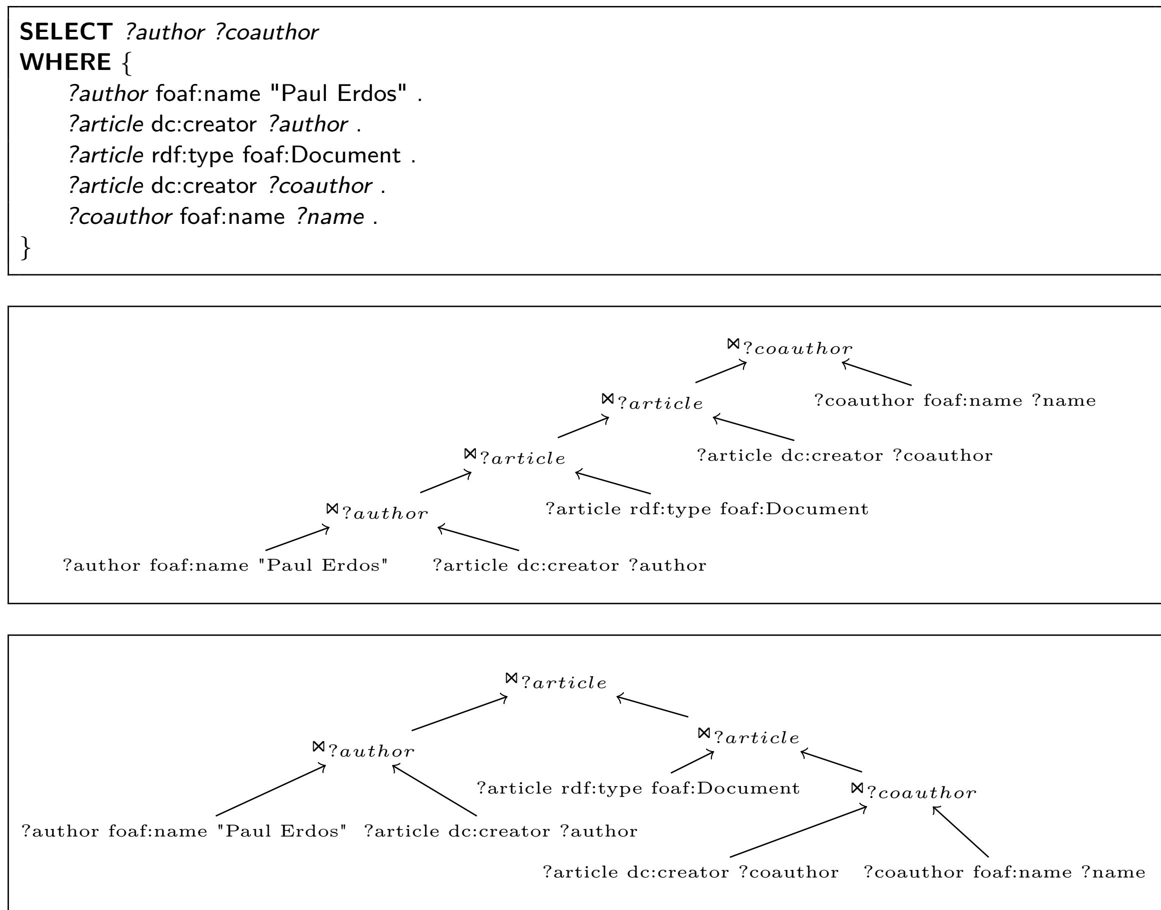


Fig. 5.1. *Left-deep tree* (above) and a *bushy tree* (below) representation for a SPARQL query with five triple patterns grouped in a basic graph pattern.

Left-deep Trees define a chain of binary operators, i. e. starting with two input relations (triple patterns in SPARQL), the result of each join operation is joined with the next input relation. Thus, all query operators are processed in a pipeline, which basically means that each join waits for the results of the left child operator. Left-deep trees are not well suited for queries with Cartesian products, unless the Cartesian product can be performed at the root node. Otherwise large intermediate results may be produced rather early. Moreover, the pipelined execution can become a bottleneck if large results have to be retrieved from different data sources. A parallel execution of triple pattern may alleviate that problem but at the cost of a higher resource consumption.

Bushy Trees support arbitrary branching of sub queries. It has been shown that bushy trees offer better query execution plans for the optimization of relational queries [121]. Moreover, the different subtrees can easily be executed in parallel. This allows for more optimization options, especially for distributed queries.

Physical Query Operators

A logical query plan defines the order in which logical query operators have to be applied to produce the final query result. Different equivalent logical query plans will produce the same result. However, a logical query plan does not specify how a query should be executed. Therefore, *physical query operators* define the actual algorithms which should be used to implement data access, e. g. table scan, or how to perform join operations, like nested-loop join, sort-merge join, or hash join. Hence, in a physical query plan logical query operators are annotated with the physical implementation that should be used to evaluate the operator.

However, in a federated database it is not possible to specify the physical access plan for a leaf node because the actual access implementations, like table scans or index lookups, are typically not known. The choice of physical join operators, however, plays an important role as they can significantly influence the performance of the query execution. The next section will give some details on the different join implementations.

5.1.2 Join Algorithms

The query execution performance depends a lot on the use of efficient join algorithms [159]. For centralized databases there exist different join implementations which take into account aspects like *memory size*, *I/O operations*, *size of input relations*, and *indexes on attributes*. Join algorithms for distributed databases also have to consider the *join site*, the *number of messages*, and the *amount of data* transferred. Following is a short overview of the most important join algorithms used in centralized and distributed databases.

Nested-Loop Join. This is the simplest join algorithm which basically iterates all join combinations of two input relations and selects only tuples matching the join condition [159]. One input relation is defined as *outer relation* and the other one as *inner relation*. Each tuple from the outer relation is matched against all tuples of the inner relation. Hence, the complexity is $\mathcal{O}(n \times m)$. Typically, both relations are processed block-oriented. If processing of the inner relation requires I/O activity in each loop, it is more efficient if the outer relation is the smaller relation. The nested-loop join is not suitable for joining large relations with low join selectivity but great for parallelization as block comparison can be split up easily.

Sort-Merge Join. The objective of the Sort-Merge Join [159] is to to reduce the number of tuple comparisons. It operates in two stages. First, the relations are sorted by the join attribute. Then they are scanned and compared in the order of the join attribute. The comparison becomes more expensive if the same join attribute occurs in multiple tuples, especially if they do not fit into memory. The relations are only scanned once. Hence, the complexity is $\mathcal{O}(m + n)$. The execution time depends mainly on the sorting time which is $\mathcal{O}(n \log n)$. The Sort-Merge Join is simplified if the join attributes are indexed because index scan and the actual joining of tuples can be separated.

Hash Join. The cost for comparing tuples from the first relation with tuples from the second relation is reduced with the application of hashing [159]. First, all tuples of one relation are hashed on the join attribute. Then each tuple of the other relation is compared against the limited set of the first relation. Thus, tuple combinations are ignored which cannot join. Hash collisions are problematic as they require additional comparisons. Ideally, the first relation contains the fewest distinct join attributes. Practically, the smallest relation is used. The complexity of the Hash Join is $\mathcal{O}(m+n)$ as it requires only one scan of each relation. The overall performance depends on the performance of the hash function.

In distributed databases [132] join algorithms have to be adapted for processing relations which are located at different data sources, i. e. distinct nodes in the network. There are different factors which influence the overall performance and complexity for query optimization, e. g. the *join site*. Three options are possible, i. e. the join is performed at the first node, at the second node, or at a third node which retrieves both input relations from the other two nodes. There are two approaches for transmitting the tuples of a relation from one node to the other. *Ship whole* transfers a complete relation while *fetch matches* requests only a subset of tuples with specific bindings for the join attribute [149]. Moreover, join processing can be done in a pipelined fashion or by materializing the input relations.

The choice of join algorithms typically also depends on the capabilities of the nodes, the cost for transmitting the data, and the cost for computing the join at the respective join site. In wide area networks the cost for transmitting data is much larger than the cost for local join processing [29]. Hence, an optimization of the data transfer cost at the expense of the local processing cost may be beneficial. Following is an overview of common join algorithms in distributed databases with a discussion of their application scenario and respective advantages and disadvantages.

Semi-Join. If two input relation are located at different nodes the Semi-Join [28] reduces the amount of data which has to be transferred to the join site. Instead of shipping a whole relation only attributes which can actually be matched by the join are sent. The first node applies a projection of the join attribute on its input relation and sends the projected values to the

second node. There, the Semi-Join is computed which reduces the amount of tuples that will be returned to the first node and used to compute the final join. The initial projection, the Semi-Join, and the final join evaluation require additional processing overhead but the performance is improved significantly in wide area networks due to less communication costs. A Semi-Join is typically applied on the relation with less distinct join attributes. If such information is not available, the smallest relation is typically used.

BloomJoin A Semi-Join variation is implemented by the *Bloom Join* [149]. It further reduces the amount of data transferred by sending a *Bloom filter* [33], i. e. a bit vector, instead of a set of projected join attributes to the second node. Each join attribute of the first relation is hashed. The hash value points to a bit in the bit vector which is set accordingly. Computing the Semi-Join only requires to hash the join attributes of the second relation (using the same hash function) and check if the respective bit in the Bloom filter is set. Due to potential collisions the reduced input relation, which is sent back to the first node, can be slightly larger than for the original Semi-Join. However, the reduction achieved by the Bloom filter outweighs this effect.

Bind Join Similar to Semi-Joins the communication cost and the query execution cost at data sources can be reduced with a Bind Join [93]. The Bind Join works as a nested-loop join and passes bindings for the join attribute from the outer relation to the data source wrapper of the inner relation. Results are then filtered based on the provided bindings. The performance of a bind join is typically improved if intermediate results are small and if indexes for the join attributes exist at the join side.

Symmetric Hash Join. Hash joins cannot produce results immediately because the inner relations has to be retrieved and hashed completely before the comparison with join attributes from the out relation can be done. The symmetric hash join [231, 114] applies the hashing on both input relations. Thus, each relation can be iterated in parallel and checked if there is a matching tuple in the other relation. This join is suitable for event-driven join processing, i. e. tuples are not fetched from the relation when needed but the arrival of a new tuple triggers the join processing. A common problem with symmetric hash joins is the increased space requirement for storing the hash tables. A solution is provided by the XJoin [220] implementation which employs external storage for tuples which do not fit into memory.

5.1.3 Query Optimization Strategies

The primary objective of query optimization strategies is to chose the best query execution plan according to specific performance criteria, i. e. usually the query execution time. However, finding the optimal query plan would require to execute and compare all possible query plans, which is obviously

infeasible. Hence, it is necessary to employ strategies which allow for finding a sufficiently good query execution plan using only the limited amount of information about data sources and query processing capabilities that is available a priori. Following is an overview of two common query optimization strategies used in (federated) relational databases.

Cost-based Query Optimization

Dynamic Programming [205] is a well-known cost-based query optimization strategy in relational databases. It employs exhaustive query plan iteration and cost estimation to find the best query execution plan. The basis for the cost computation is a cost model which is used to estimate the processing cost for each query operator depending on the size of the input relations. Statistics-based selectivity estimation allows to determine the size of intermediate results. Dynamic Programming iterates through all query plan combinations for a specified query plan tree structure (cf. Sec. 5.1.1), i. e. joins are basically employed in any possible combination. This ensures that the optimal query execution plan is found, given that the cost estimation is accurate. Since the number of overall query execution plans, which have to be compared, increases exponentially with the number of joins Dynamic Programming prunes inferior plans (i. e. equivalent plans with higher estimated cost) as soon as possible. Queries with many joins can be optimized with *Iterative Dynamic Programming* [133], an optimized approach for handling a large number of intermediate query plans. Further optimizations for efficiently generating query plans for specific join structures with Dynamic Programming have been presented by Moerkotte and Neumann [161].

Adaptive Query Optimization

Query optimization strategies which follow the *optimize-then-execute* approach assume a stable environment with sufficient statistical information about the datasets in order to find the best query execution plan. If these requirements cannot be satisfied, e. g. because of missing or limited statistics, unexpected correlation, unpredictable cost, or dynamic data, a static query optimization can easily produce ineffective query execution plans. Moreover, querying autonomous data sources introduces additional problems, like server timeouts, limited bandwidth, or other network related problems. Adaptive query processing [65] tries to overcome such problems during query execution by introducing a feedback cycle. This allows for query adaption at runtime, e. g. by altering the query execution plan or by changing the scheduling of query operators [16, 64].

5.2 Query Optimization Challenges for Federated Linked Data

There are many similarities between federated databases and the federation of RDF datasets in the Linked Data cloud. Hence, query optimization for federated Linked Data deals with almost the same challenges as encountered in distributed and federated databases [132, 206]. However, relational query optimization approaches can only be employed with certain restrictions because of the specific characteristics of RDF's graph-based data model and due to architectural differences. This section describes SPLENDID's query optimization goals, compares federated databases with the federation of Linked Data, and discusses common query optimization strategies and their adaption for federated, heterogeneous RDF data sources.

5.2.1 Optimization Objectives

Minimizing the query execution time is usually the most important objective for a query optimizer. Additionally, the processing overhead for the network communication and the amount of data transferred over the network are relevant in a distributed setting. The SPLENDID query optimizer utilizes common cost-based query optimization techniques from distributed databases and allows for the transparent federation of Linked Data sources. Hence, the objective is to adapt and integrate efficient join implementations and generate optimal query plans which are suitable for parallel query execution. The result of the query optimization is a query plan which is optimized according to following aspects.

Result Completeness

Precision and recall are typical performance criteria in classic information retrieval systems [17]. But they are not directly applicable for the evaluation of SPARQL queries on RDF data because graph pattern matching in SPARQL produces exact results. Moreover, a ranking of results is only possible if an explicit sort order has been specified. Therefore, top-k retrieval [151] and relevancy-based performance criteria are not generally applicable. Instead, the objective of SPLENDID's query optimization strategy is to return complete results.

Minimizing Network Communication

Distributed query processing requires communication with different data sources and exchanging queries and results over the network. The cost for establishing network connections, sending requests and retrieving results in a wide area network, is significantly higher than the cost for local query processing [29]. Moreover, the number of requests and the amount of data which

is transferred over the network has a direct influence on the query execution time. Hence, the query processing performance can be greatly improved through the minimization of the network communication cost. Especially for a large number of data sources, like in the Linked Data cloud, the communication overhead cannot be neglected. Following is a overview of the most important aspects.

Parallel Execution. The source selection (cf. Chapter 4) may identify multiple data sources which can return results for the same part of a query. In this case, a sub query can be executed in parallel at different data sources. Parallelization can also be done for different parts of a query if no sequential order has to be asserted. The applicability of parallel execution depends on the structure of a query, the query processing capabilities of SPARQL endpoints, and the join implementations which may require that intermediate results have to be retrieved completely before the next query operator can be executed. Parallelization is possible in general if results can be retrieved independently from different data sources. Sequential query execution is less efficient because the overall query execution time is the sum of the execution times of all requests to individual data sources. However, there is also a limit for parallelization as only a maximum number of connections can usually be open at the same time for multiple data sources.

Location of Join Processing. SPARQL endpoints are only capable of answering queries on their provided data but cannot exchange meta data between each other. Therefore, the join computation for intermediate results from different data sources usually has to be done at the federator, i. e. the query issuer. The query processing overhead for the federator can be reduced if queries are split into larger sub queries, such that each SPARQL endpoint computes join results for complete sub queries, or if semi-joins can be applied. Choosing different join sites is a suitable approach for better load balancing and to minimize the processing cost for the federator or different SPARQL endpoints.

Stream Processing. Transferring large result sets over the network can delay the query execution significantly. Thus, the query execution can be sped up when consecutive query operators can already be applied on the data stream while the data is still being retrieved, which allows for returning results as soon as they become available. However, such an approach may be problematic when an order has to be applied on the results. In addition, stream processing is only possible for SPARQL result sets. There is currently no support for stream-based processing of SPARQL queries with bindings. Stream processing can also help to cope with unpredictable data rates. But it cannot avoid situations where unresponsive SPARQL endpoints become the bottleneck in the query execution.

5.2.2 Comparison of Linked Data with Federated Databases

As pointed out earlier, there are several differences between a federation of Linked Data sources and federated databases. Following is a detailed comparison of these differences which lead to certain constraints for the adaption of database technology for the query optimization on federated Linked Data.

Data Integration. Federated databases [206] use custom wrappers to integrate diverse data sources with different schemas and capabilities. A wrapper provides different functionalities, e. g. schema mapping, translation of algebraic query expression to specific database implementations (i. e. table scans, index scans, join implementations), support for query planning with information about the database content including statistical information, and cardinality estimates for cost-based query optimization. Linked Data supports URI resolution, provides SPARQL endpoints, and represents all data with RDF. There is no need for specific data source wrappers, as the data format and data access is already standardized. However, a SPARQL endpoint typically does not provide additional meta data, like the supported SPARQL version.

Query Planning. Data source wrappers in federated databases can support query planning with information about physical operators, query plan translation, and query plan cost estimations. This helps the mediator to assess if a query plan can benefit from specific join implementations or if shipping of whole relations is necessary. Distributed query optimization for federated Linked Data has no such information about the different data sources. Implementation details of the underlying triple stores are usually hidden. Thus, a federator can only assume that all data sources have common capabilities and provide the same set of features.

Data Structure. Relational data is highly structured. RDF in turn is less structured and usually comes without explicit schema information. Moreover, theres exist different types of links between the resources of different Linked Data sources. Query optimization in (federated) relational databases often relies on the *attribute independence* and *uniform distribution* assumption. However, since RDF data is highly correlated common estimation approaches from relational databases [205] cannot be applied.

Meta-Data and Statistics. Data source wrappers in federated databases maintain meta-data about data sources, including information about supported operator implementation and the availability of indexes. Additionally, relational database employ sophisticated statistics, like histograms [120] for attributes, in order to allow for accurate cost estimation. Such statistical data can also be available within data source wrappers to be used for query planning. In contrast, statistics for Linked Data are hardly available and usually do not contain detailed information as histograms, but mainly basic counts for the schema data. However, due to the high correlation of RDF resources a sophisticated cost-based query optimizer would need more detailed statistical data than for relational databases.

5.2.3 Distributed SPARQL Query Optimization Strategies

Finding the optimal query execution plan, whether for federated databases or federated Linked Data, is challenging because various factors influence the overall query execution time. Existing query optimization strategies use heuristics, statistics-based cost estimates, or adaptive query optimization in order to determine the best possible query execution plan with respect to available information about the datasets and capabilities of the involved data sources. Sophisticated query optimization strategies can generally achieve better results than heuristics. But they have a higher computational overhead and may require specific meta data and statistics, e. g. for estimating and comparing the cost of different query execution plans. Moreover, when using the *optimize-then-execute* approach the time required for generating the optimized query plan has to be considered within the overall query execution time, i. e. a query optimization strategy can be inefficient if finding the optimal query plan for a complex query takes much longer than choosing and executing a less optimal query plan.

Join order optimization [159, 179] plays an important role, especially for distributed SPARQL queries because the size of intermediate results has a significant influence on the query processing cost and on the amount of data transferred over the network. Many SPARQL query optimizations strategies rely on heuristics for optimizing basic graph patterns. But also cost-based and adaptive SPARQL query optimization strategies can be used, e. g. if specific statistics about the Linked Data sources are available. Advantages and disadvantages of these three approaches will be outlined in the following. For a general survey of state-of-the-art Linked Data federation approaches the interested reader is referred to [190]. More detailed investigations concerning the complexity of SPARQL query optimization have been conducted by Pérez et al. [183] and Schmidt et al. [200].

Heuristic-based Query Optimization

Heuristic-based query optimization is often used when information about the data sources is limited or not available at all. The general idea of all heuristic-based query optimization approaches is to order a query's join operators such that small intermediate result sets are produced first, i. e. to start with the most selective query expressions. However, the main problem for SPARQL queries is to determine the actual selectivity of individual triple patterns and of joins only based on the syntax and structure of the basic graph patterns. *Variable counting* [208, 203] is a commonly used technique, i. e. triple patterns with two bound variables are considered more selective than triple patterns with one or no bound variables. In addition, the position of bound variables plays also an important role. The selectivity of triple patterns with a bound subject is considered higher than with a bound object, or with a bound predicate. This assumption is typically supported by the characteristics of RDF

graphs which contain few distinct predicates but many different resources. Selectivity estimation for joins is even more complicated. Again, the position of join variables is important. Subject-subject joins are more common than subject-object joins or object-objects joins. Hence, joins are executed in this precedence order. Rare joins, e. g. involving predicates, are considered highly selective. The maximum selectivity is applied for triple patterns where the join variable is bound to a specific value whereas joins over triple patterns with no common join variable have the lowest selectivity.

There exist several heuristics-based SPARQL query optimization implementations focusing on join order optimization of basic graph patterns in a centralized or distributed setting. Stocker et al. [208] use heuristics in combination with a probabilistic model based on a central triple store with precomputed statistical data. Tsialiamanis et al. [216] favor heuristics because they argue that cost-based SPARQL query optimization is generally complicated because of the high data correlation in RDF data which is rarely considered in relational optimization approaches. Their approach tries to maximize the number of merge joins by looking at structural characteristics of the join graph, thus, reducing the problem to finding the maximum weight independent set. FedX [203] implements heuristics for distributed SPARQL query optimization using a variation of the variable counting technique of [208] with a preference to sub queries which can be evaluated at the same data source. Montoya et al. [163] use heuristics for creating sub queries with triple patterns which can be executed on the same data source. Each sub-query can be executed in parallel. The query planner uses a greedy algorithm to combine all sub queries in a bushy tree while avoiding Cartesian products and minimizing the height of the query operator tree.

The application of heuristics is often sufficient for the optimization of basic SPARQL queries. But complex query expressions with many joins and additional constraints, like filters and optional parts, are much more challenging. Moreover, the join order optimization with heuristics typically leads to a pipelined execution which is not optimal for the distributed scenario of federated Linked Data where parallel execution can be highly beneficial.

Cost-based Query Optimization

Cost-based query optimization is a typical optimization approach in relational databases and allows for accurate estimations of the size of intermediate results given that relevant statistical information is available. However, it is not widely used for optimizing SPARQL queries yet as it requires sophisticated statistical information about the correlations found in RDF data. There are two promising research implementations for centralized RDF triple stores, namely RDF3X [172] and Hexastore [226]. RDF3X provides very efficient query optimizations using dynamic programming and many different indexes on all possible variations of bound variables in triple patterns and join combinations.

Hexastore applies similar optimization techniques but with a slightly different RDF indexing approach.

The application of cost-based query optimization for distributed SPARQL query processing is currently limited. So far, only two federation systems, i. e. DARQ [188] and SPLENDID [88], implement distributed query optimization with cost-based query optimization (dynamic programming) and have been evaluated with real distributed RDF data. However, DARQ is not scalable beyond a few datasets as it requires the manual definition of so called capabilities for each data source. SPLENDID's goal is to allow a more flexible federation of Linked Data sources using automatically created statistical data.

Adaptive Query Optimization

Query optimization for autonomous data sources can be challenging due to unpredictable network conditions. Adaptive query processing tries to cope with these problems by re-optimizing queries during execution. Research on adaptive query processing for Linked Data federation has not received much attention until recently. For example, Anapsid [6] deals with unreliable data rates and interrupted result retrieval by employing the flexible XJoin [220] operator. In contrast, Avalanche [22] selects top-k query execution plans which are processed in parallel. The results of the fastest query execution plan are returned to the user. A combination of cost-based query optimization with adaption heuristics is presented by Aderis [147]. However, it lacks a detailed description of the adaptive join re-ordering and a suitable evaluation. Finally, Hartig [103] proposed link traversal query processing (cf. Sec. 3.5) with iterators that can detect blocking of result transmission and issue new requests to obtain results from alive data sources. Although the latter approach is well suited for querying interlinked RDF datasets it cannot easily scale up to the size of the Linked Data cloud.

5.2.4 Maintaining Linked Data Statistics

Cost-based query optimization techniques generally produce better results than heuristics. But they needs statistical information in order to estimate the cost for the query execution plans. With respect to the aforementioned characteristics of federated Linked Data sources there are different challenges for maintaining such statistics. Especially due to the loose structure and the correlations in RDF data detailed statistics are required in order to obtain accurate cost estimations for query plans.

The generation of statistical information for a data set requires a certain effort which is necessary before any cost-based query optimization can be done. However, Linked Data providers typically do not consider this effort beneficial, which means that there exist only few Linked Data sources which have statistical data available. Moreover, these statistics vary a lot in their quality and the provided level of detail is often not sufficient. Therefore, it

is usually necessary to have a pre-processing step before setting up a Linked Data federation in order to collect, generate, and store the statistical data for all known data sources. Certain statistics can be easily retrieved through specific aggregate queries on SPARQL endpoints. However, such an approach requires a number of initial query requests before all required statistics are available for the query optimization. Thus, it is not scalable for a federation of the Linked Data cloud.

The management of the statistics has to be done by the federator, typically in specific indexes, because the query optimizer needs to access them frequently. However, the indexes can become quite large due to the size of detailed statistics for many Linked Data sources. This can be a serious problem, especially since the statistics should usually be kept in memory to allow for fast lookups. Hence, the main challenge is to find the optimal trade-off between the maintained amount of statistical information and the achievable accuracy of the cost estimation for query plans. Figure 5.2 illustrates this trade-off and shows the potential best threshold with respect to the statistics maintenance cost. While maintaining more statistical data becomes more expensive the benefits for the cardinality estimation will decrease. In fact, after a certain point further improvements of the accuracy may not even have any more effects on the order of operators in a query plan.

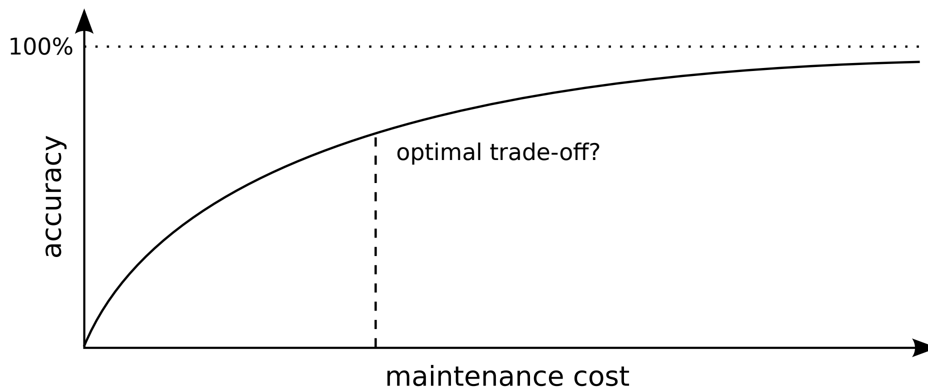


Fig. 5.2. Trade-off between statistics maintenance cost and cardinality estimation accuracy for complex query plans

5.3 Join Order Optimization for Distributed SPARQL Queries

The objective of SPLENDID is to efficiently optimize distributed SPARQL queries for federated Linked Data sources by adapting cost-based query optimization techniques known from federated databases which can produce better query execution plans than common heuristics-based join order optimization approaches. Following, SPLENDID's query optimization approach, with solutions for the aforementioned challenges, will be explained in detail.

5.3.1 Linked Data Integration Requirements

The integration of different Linked Data sources is generally very easy due to the use of RDF and SPARQL endpoints as common data format and query interface, respectively. However, SPARQL endpoints may support different versions of SPARQL or offer proprietary functions which are specific to the underlying triple store implementation. It is often hard to find out the exactly supported feature set for each SPARQL endpoint. Moreover, it is infeasible to implement a federation of Linked Data sources which uses a customized query execution for each SPARQL endpoint according to the supported feature set. Therefore, SPLENDID relies on SPARQL 1.0 [187] for the distributed query execution as it cannot be assumed that SPARQL 1.1 [98] is supported by every SPARQL endpoint

Cost-based query optimization requires statistical information about the content of the data sources. These statistics should be available in a standard format, preferable in RDF, which can be easily processed. The use of RDF allows for directly querying statistical data with the same mechanisms as for querying the actual RDF data. Moreover, there is no need for different data formats or local data structures for maintaining the statistics. Statistical information can be expressed in RDF with different vocabularies, like *Scovo* [108] or the *RDF Data Cube Vocabulary* [59]. But these RDF vocabularies are designed for expressing very generic statistics, e. g. with multiple dimensions and various data types. Hence, they are quite verbose and not really suited for the use in a Linked Data scenario. Another suitable choice is VOID, i. e. the *Vocabulary for Interlinked Datasets* [9]. VOID is primarily designed for describing general meta-data of Linked Data sources but also allows for expressing basic statistical data. In an early draft version VOID relied on the Scovo vocabulary, which was later replaced in favor of a more concise representation. Currently, only basic statistical data can be expressed with the VOID vocabulary. However, it is possible to easily extend VOID with additional statistical data, e. g. using other RDF vocabularies. VOID descriptions can be generated with automatic tools and a serialization to the Turtle RDF format is still human-readable (cf. Fig. 5.4). Moreover, VOID has been employed for SPLENDID's source selection (cf. Chapter 4).

5.3.2 SPARQL Join Implementations

The efficiency of specific physical join implementations depends on different aspects, like data location, input data size, and join selectivity. Especially for distributed data sources it makes a huge difference how much data has to be transmitted over the network. Section 5.1.2 described different join implementation which are employed in federated databases. They are generally applicable for distributed SPARQL query processing as well, e. g. semi-join and bind join, which significantly reduce the network communication cost.

But there are also restrictions of the SPARQL syntax [187] (e. g. result bindings only supported by SPARQL 1.1) and the SPARQL protocol [55] (e. g. lacks streaming of result bindings) which renders such join implementations for SPARQL endpoints less straight forward.

Semi-Join

A Semi-Join [28] applies a projection on the join attributes and two consecutive joins on partial data in oder to reduce the size of intermediate results sets which have to be transferred over the network. These three operations can be easily expressed with SPARQL but the transmission of the partial data is complicated, because SPARQL 1.0 does not allow for including variable bindings in a query. This was fixed with the SPARQL 1.1 specification [98] by introduced the new keyword `VALUES` [186] (`BINDINGS` in the former draft versions) which allows to include values for tuple bindings in a SPARQL 1.1 query. As of March 2013 SPARQL 1.1 has become a "W3C Recommendation" but it cannot be assumed that all existing SPARQL endpoints already support it.

Workarounds for implementing Semi-Joins with SPARQL 1.0 can be found in Networked Graphs [196], Distributed SPARQL [233], DisMed [166], and FedX [203]. One possibility is to define variable restrictions with FILTER expressions. But including a long list of filtered values also yields a longer query and the evaluation of many filter expressions is usually not very efficient. Instead, it is better to use a UNION expression, which includes the graph pattern multiple times, each having the variable substituted with one value binding. Although the UNION approach increases the query length as well it can usually be evaluated more efficiently. The drawback, however, is that the variable is eliminated and the substituted value bindings cannot be returned in the result set. Therefore, FedX [203] implements an extension of variable names with unique identifiers in order to retain the original value bindings. For example, given n variable bindings $\mathcal{Y}_1 \ldots \mathcal{Y}_n$ for variable $?y$ in triple pattern '$?x$ dc:creator $?y$'. FedX will rename the unbound variable $?x$ in the rewritten triple pattern for each binding $\mathcal{Y}_i$ by adding a unique suffix '_i' and storing the mapping ($?x$_i, $\mathcal{Y}_i$). All query pattern variations are shown in Fig. 5.3.

Bind-Join

The problem with executing semi-joins on SPARQL endpoints is that the complete SPARQL query has to be parsed before the actual query execution starts. Hence, a large number of variable bindings, which may be received slowly from one data source and need to be transmitted entirely to the other data source, can significantly delay the overall query execution. The bind join [93] (cf. Sec. 5.1.2) instead, sends query and variable bindings separately. Thus, the data source can optimize the query plan first and produce results as soon as new variable bindings are available. However, SPARQL does not support such a separation, i. e. the bindings have to be included in the SPARQL query.

SPARQL 1.1 Expression:

```
1  SELECT ?article ?author ?coauthor
2  WHERE {
3    ?article dc:creator ?author.
4    ?article dc:creator ?coauthor.
5  }
6  VALUES (?author ?coauthor) {
7    ("Paul Erdos" "Andreas Blass")
8    ("Paul Erdos" "Walter Deuber")
9
10 }
```

Filter Expression:

```
1  SELECT ?article ?author ?coauthor
2  WHERE {
3    ?article dc:creator ?author.
4    ?article dc:creator ?coauthor.
5    FILTER (
6    (?author = "Paul Erdos" &&
7     ?coauthor = "Andreas Blass") ||
8    (?author = "Paul Erdos" &&
9     ?coauthor = "Walter Deuber"))
10 }
```

UNION Expression:

```
1  SELECT ?article
2  WHERE {
3    {
4      ?article dc:creator "Paul Erdos".
5      ?article dc:creator "Andreas Blass".
6    } UNION {
7      ?article dc:creator "Paul Erdos".
8      ?article dc:creator "Walter Deuber".
9    }
10 }
```

FedX UNION Expression:

```
1  SELECT ?article_1 ?article_2
2  WHERE {
3    {
4      ?article_1 dc:creator "Paul Erdos".
5      ?article_1 dc:creator "Andreas Blass".
6    } UNION {
7      ?article_2 dc:creator "Paul Erdos".
8      ?article_2 dc:creator "Walter Deuber".
9    }
10 }
```

Fig. 5.3. Variable bindings for Semi-Join: using `VALUES` syntax in SPARQL 1.1 and workaround solutions for SPARQL 1.0 with FILTER and UNION expressions

Therefore, a bind join can only be imitated to some extend. For example, several distributed SPARQL query processing implementations rely on a nested loop join which sends a SPARQL query for each binding. But this is very inefficient if the outer relation is large and many individual bindings have to be transmitted. In order to reduce the number of requests, a block-wise transmission of bindings can be used. FedX [203] implements a "blocked bind join" which employs the UNION-based Semi-Join approach described above to send queries containing "blocks" of bindings.

Non-Blocking Local Joins

Local joins operate on locally available data or the results retrieved from remote SPARQL endpoints. Common join implementation may not be efficient as the network communication cost can significantly hamper the overall performance, e. g. because nested-loop joins first have to process the inner relation before the join processing can start. Under these conditions it is better to use an event-based query processing approach, i. e. result tuples are handed to the next operator when they become available. Thus, a join operator should be able to process result tuples in a non-blocking way and independently of the input relation that produces them. There exist a few such implementations for SPARQL. Ladwig and Than [135] use a parallel hash join. It was further

extended to the SIHJoin [136], a recently proposed non-blocking join operator similar to the XJoin [220], which was designed to join remote data with locally stored tuples. Hartig et al. [104] implement a Non-Blocking Iterator Join for the link traversal query processing (cf. Sec. 3.5). Result tuples are fetched and if no match can be computed the tuple will be put in a queue for later consumption.

Support of Join Implementations in Query Models

Most RDF Federation implementations, like DARQ [188] and SemWIQ [140], which are based on the Sesame [39] or Jena2 [230] RDF framework, employ usually only one join type in their query plans, i. e. a nested-loop join. This limitation has several disadvantages with respect to the efficient execution of distributed queries. First, nested-loop joins are highly inefficient for joining large intermediate result sets from two different data sources, because a large number of messages need to be exchanged. Furthermore, one join implementation does not fit all conditions. Depending on the size and location of the input data, different join implementations may be chosen. However, the query model has to support the use of different join implementations. Typically, the join nodes in an abstract syntax tree need to be annotated with the desired physical join operator. SPLENDID employs the Sesame Query Model which had to be extended to support this kind of annotation for different physical join implementations. Currently two different join algorithms are supported in SPLENDID, i. e. a Bind Join similar to FedX's Controlled Bound Join and a Parallel Hash Join. Hence, the query planning is more flexible and joins can be chosen appropriately.

As soon as SPARQL 1.1 becomes widely supported it will also be possible to develop more effective query optimization techniques for Linked Data federation. For example, the introduction of the `SERVICE` keyword allows for defining remote request to other SPARQL endpoints within a query. Thus, a cooperative processing of sub queries by different data sources may be possible if such a query is sent to one SPARQL endpoint which is capable of forwarding the inner query to a different SPARQL endpoint and then include the returned results in its own query evaluation. Such an approach would relieve the federator because join processing can be delegated to other SPARQL endpoints. However, the complexity of such a query optimization approach is significantly higher due to additional optimization parameters, e. g. query shipping versus data shipping and load distribution among the involved endpoints.

5.3.3 Cost-based Join Order Optimization

The core problem for the SPARQL query optimization strategies is to find the best execution order for the conjunctive joins defined in basic graph patterns, i. e. a set of triple patterns with optional filter expressions. The formal model

of Hartig and Heese [106] defines query operators based on the consumed input data and the produced output data, i. e. data flows of tuples with bindings for specific variables. The challenge for the query optimization is to find the optimal order of join operators such that the input data of each join operator contains bindings for the required input variables and that the overall processing cost is minimized.

In contrast to heuristics-based approaches [106, 203] SPLENDID employs *dynamic programming* [205] for cost-based join order optimization. Dynamic programming performs an exhaustive search on all possible combinations of joins in a query plan. Pruning techniques are applied to reduce the search space and prevent an exponential growth in the number of query plans. The dynamic programming algorithm initially starts with all *access plans* (i. e. all sub queries) and creates iteratively new query plans with $2..n$ operators by joining query plans produced in the previous iterations. Each new join combination is created for all supported join types, i. e. parallel hash join and blocked bound join in SPLENDID. At the end of each iteration all query plans are pruned such that for equivalent query plans only the one with the lowest cost is retained for further query combinations.

Algorithm 2 shows the pseudo code for the dynamic programming algorithm used in SPLENDID. The result is the optimal query execution plan, i. e. an operator tree annotated with the chosen join implementations and overall lowest estimated execution cost. The algorithm starts with the set of sub queries $\{Q_i, \ldots, Q_n\}$ which are obtained from the source selection. Federated databases may apply different access plans for evaluating a sub query on a

Algorithm 2 Query Optimization with Dynamic Programming

Input: a set of sub queries $\{Q_1, .., Q_n\}$ = sourceSelection(Q)
Output: an optimal query execution plan for Q
1: // create initial access plans, i. e. a SPARQL request for each sub query
2: **for** $i = 1 \to n$ **do**
3:    optPlan($\{Q_i\}$) = {SparqlRequest(Q_i)}
4: **end for**
5: // iterate over all join combinations of the sub queries
6: **for** $i = 2 \to n$ **do**
7:    // assess each subset of i joined sub queries
8:    **for all** $\mathcal{Q} \subseteq \{Q_1, \ldots, Q_n\}$ such that $|\mathcal{Q}| = i$ **do**
9:       optPlan($\mathcal{Q}$) = $\emptyset$
10:       // create all join variations for $\mathcal{Q}$ using query plans of previous iterations
11:       **for all** $O \subset \mathcal{Q}$ **do**
12:          optPlan($\mathcal{Q}$) = optPlan($\mathcal{Q}$) $\cup$ joinPlans(optPlan(O), optPlan($\mathcal{Q} \setminus O$)))
13:          prunePlans(optPlan($\mathcal{Q}$))
14:       **end for**
15:    **end for**
16: **end for**
17: prunePlans(optPlan($\{Q_1, \ldots, Q_n\}$))

data source but in SPLENDID there is only one type of access plan, i. e. a standard SPARQL SELECT query. Hence, in the first iteration step (lines 2-4) each sub query is mapped to exactly one SPARQL request which is added to the map *optPlan*. In the following iterations (lines 6-16) new query plans are generated by adding one join at a time (cf. condition in line 8), thus, producing two-way, three-way, ..., n-way joins on the given sub queries. New query plans are obtained by combining all permutations of two query plans from previous iterations (line 11). For example, a four-way join can be made up of two two-way joins or a three-way join with a sub query. Since SPLENDID employs two different join implementations, $joinPlans()$ yields a set of alternative query plans which are added to *optPlan* for the respective set of sub queries $\mathcal{Q}$ (line 12). At the end of each iteration (line 13) inferior query plans are removed with $prunePlans()$, i. e. for logically equivalent query plans only the query plan with the lowest cost will be maintained. Finally, *optPlan* contains the best execution plan for joining all sub queries $\{Q_i, \ldots, Q_n\}$.

Sub queries in SPLENDID are either exclusive groups, i. e. a set of triple patterns which are evaluated at one data source, or they contain just a single triple pattern which may be evaluated at different data sources. In the latter case, a union will be used to aggregate the results of requests to multiple data sources. Additionally, the query plan generation prefers joins of sub queries with a common join variable. Thus, Cartesian products are only considered if no other join combination is possible. The algorithm implemented by SPLENDID also allows for including filter expressions which are applied to a query plan at the end of an iteration step if the query plan contains all variables of the filter expression.

5.3.4 Representing RDF Statistics with VOID

The motivation for using VOID descriptions to manage statistical information about Linked Data sources was already given in Sec. 5.3.1. Figure 5.4 shows an excerpt from the description of the Drugbank dataset. The first section (lines 6-10) provides general statistics, e. g. item counts for the the overall number of triples, properties (predicates), entities, and the distinct number of subjects and objects. Thereafter follows schema-specific data (lines 13-53) concerning the number of RDF classes and predicates. The final section (lines 55-60) contains information about links to other datasets.

Schema Statistics The schema-specific statistics are divided into two parts. The *class partition* (`void:ClassPartition`) gives details about all RDF classes, i. e. resources which occur in object position of an RDF triple with the predicate `rdf:type`. Information about predicates is contained in the *property partition* (`void:PropertyPartition`). A class partition references the class with `void:class` and specifies the overall number of entities of this class by `void:entities`. A predicate in a property partition is identified via `void:property` and defines the respective number of occurrences in all RDF triples of the dataset with `void:triples`.

```
 1  :Drugbank a void:Dataset ;
 2
 3  dcterms:title "Drugbank" ;
 4  void:sparqlEndpoint <http://...> ;
 5
 6  void:triples "766920" ;
 7  void:entities "19693" ;
 8  void:properties "119" ;
 9  void:distinctSubjects "19693" ;
10  void:distinctObjects "276142" ;
11
12
13  void:classPartition [
14          void:class rdf:Property ;
15          void:entities "117"
16  ] , [
17          void:class rdfs:Class ;
18          void:entities "6"
19  ] , [
20          void:class drugbank:drug_interactions ;
21          void:entities "10153"
22  ] , [
23          void:class drugbank:drugs ;
24          void:entities "4772"
25  ] , [
26          void:class drugbank:enzymes ;
27          void:entities "53"
28  ] , [
29      ...
30  ] ;
31  void:propertyPartition [
32          void:property rdf:type ;
33          void:triples "24522" ;
34          void:distinctSubjects "19693" ;
35          void:distinctObjects "8"
36  ] , [
37          void:property rdfs:label ;
38          void:triples "19627" ;
39          void:distinctSubjects "19570" ;
40          void:distinctObjects "18780"
41  ] , [
42          void:property owl:sameAs ;
43          void:triples "9521" ;
44          void:distinctSubjects "2209" ;
45          void:distinctObjects "9486"
46  ] , [
47          void:property drugbank:absorption ;
48          void:triples "975" ;
49          void:distinctSubjects "975" ;
50          void:distinctObjects "868"
51  ] , [
52      ...
53  ] .
54
55  :SameAsLinks a void:Linkset;
56      void:subset :Drugbank ;
57      void:target :Drugbank ;
58      void:target :DBpedia ;
59      void:linkPredicate owl:sameAs ;
60      void:triples "9521" ;
```

Fig. 5.4. VOID statistics for the Drugbank dataset

Entity Statistics Detailed information about RDF entities cannot be included in VOID statistics because of the huge amount of data. Nevertheless, with `void:distinctSubject` and `void:distinctObject` there is at least some entity-related data which can be used for SPLENDID's cost estimation. However, only rough estimates can be computed based on these two values. Therefore, the VOID descriptions have been extended with distinct subject and object counts for each predicate.

Link Statistics VOID also allows for describing links between datasets. This is done with *linksets* which are a subclass of datasets. An example is given in Fig. 5.4 in lines 55 to 60. Each linkset has two target datasets specified by `void:target` or `void:subjectTarget` and `void:objectTarget`. The predicate and the number of triples representing the links is defined with `void:linkPredicate` and `void:triples`, respectively. The property `void:subset` defines in which dataset the links can be found. This reflects the fact that links between datasets can also be defined in third party datasets.

The statistical data of the VOID descriptions is used to estimate the result size of query expressions. This information is required for the cost computation of query execution plans which will be described in the next section.

5.3.5 A Cost Model for Distributed SPARQL Queries

Relational databases use cost models to compute an estimated execution time for each query plan. A cost model basically considers *CPU cost*, *I/O cost*, and for distributed queries also *communication cost* [153]. The cost computation is done recursively, i. e. the cost of a query operator also includes the cost for all operators in its sub tree. Hence, the total cost of a query plan is identical to the cost of the root node. Calculating CPU and I/O cost depends on the implementation of the physical operators. Since the cost for evaluating an operator basically depends on the size of the intermediate results the cost formulas are parameterized by the cardinality of the input relations. An adaption of this approach for SPARQL queries is easy due to the algebraic similarity of SPARQL and SQL, e. g. Obermaier and Nixon [177] present a formalization for cost estimation of basic SPARQL query operators which derives its cost formulas from cost estimations of equivalent query operators in relational databases.

Remote Queries

An important aspect for distributed query execution in federated systems is the cost estimation for query parts which are executed at remote data sources [194]. Usually, the federator has no knowledge about the physical data layout or the query processing capabilities of a distributed data source. Data wrappers in federated databases may be able to provide source-specific cost estimates for a given query but for SPARQL endpoints it is infeasible to obtain such information. However, the local query execution time is commonly negligible for distributed query processing in wide area networks, like the Linked Data cloud, because the communication cost is the dominating factor [29]. Therefore, SPLENDID considers only the communication cost in its cost model. An extension with respect to local CPU and I/O cost at the federator, however, is possible.

Communication Cost

The communication cost is typically based on two factors, namely the number of messages sent and the amount of data transferred [149]. Messages are sent whenever a SPARQL query or a result set is transmitted between the federator and a SPARQL endpoint. The message size of SPARQL queries and result sets varies. But for simplification of the computational model it is assumed that a SPARQL query always fits into one TCP/IP data packet and that results tuples have an average size which can be used to estimate the overall size of a response for a known number of contained result tuples.

Data Transfer Cost for Physical Joins

The use of specific join algorithms, like semi-join or bind join, has a significant influence on the amount of data transferred over the network. Hence, different cost formulas are used for each join implementation. The join cost depends on the way the data is retrieved by the federator, i. e. a local join of the results of two SPARQL requests (*ship-whole*), or a bind join which substitutes values of the first result set as bindings in the query for the second query expression (*fetch-as-needed*). SPLENDID relies on this distinction for computing the communication cost for the employed hash join and bind join and applies the join cost definition of DARQ [188]:

$$\text{Hash Join:} \quad c(q_1 \bowtie_H q_2) = |R(q_1)| \cdot c_{tuple} + |R(q_2)| \cdot c_{tuple} + 2 \cdot c_{query}$$
$$\text{Bind Join:} \quad c(q_1 \bowtie_B q_2) = |R(q_1)| \cdot c_{tuple} + |R(q_1)| \cdot c_{query} + |R(q_2')| \cdot c_{tuple}$$

The cost for sending a query and receiving a tuple is specified by the constants c_{query} and c_{tuple}, respectively. $|R(q)|$ defines the (estimated) cardinality of the result set of a query q. $|R(q_2')|$ in the bind join refers to the result size for q_2 with the variables bound to the values obtained from the result set of q_1. The approximation of $|R(q_2')|$ with $|R(q_1 \bowtie q_2)|$ yields a lower bound because the actual result set can be larger due to duplicates.

Cost Model Refinements

The DARQ inspired cost model assumes that sub queries are exclusive groups, i. e. a sub query is sent to a single SPARQL endpoint. In fact, a sub query may be sent to multiple SPARQL endpoints. This has no influence on the cardinality of the result set but on the overall number of SPARQL requests. Hence, the computation of the cost for requests needs a slight adaption in order to incorporate the number of assigned data sources per sub query.

Another problem is that the children of a join operator are not necessarily sub queries but other joins or filter expressions. Thus their results set is already locally available and no transfer cost has to be computed. There are essentially three variations for a join operator, it receives result tuples from (1) two sub queries, (2) just one sub query, or (3) no sub query at all. Depending on the actual situation, the data transfer cost can only be computed for the respective sub query. Otherwise the join is computed locally and there is no relevant communication cost involved in it. However, an exception is the bind join which forwards tuples as bindings to the right sub operator. If it is not a sub query, the actual data transfer cost is unpredictable, as the bindings may be used for different sub queries.

FedX's [203] blocked bound join, which SPLENDID uses for the bind join optimization, combines multiple tuple values in a single request. The reduction in the number of messages has to be accounted for in the cost formula by a constant which represents the block size. Moreover, a large block size also

increases the query size which may not fit into one TCP/IP data packet any more. For a semi-join like implementation with a large number of bindings in a query the cost for the request has to include the number of tuples from the first result set in the formula.

Finally, since the estimated query execution cost typically represents the expected query execution time it makes sense to consider the parallel execution of query operators as well. However, due to the complexity of estimating the effects of a parallel query execution correctly SPLENDID does not include any specific adjustments. Instead, the calculated cost for a query plan represents the upper bound as is executed in a pipelined fashion.

5.4 Cardinality Estimation for Distributed SPARQL Queries

The computation of the execution cost for a query plan requires knowledge about the cardinality of intermediate results sets. However, since such information is not readily available it has to be estimated with the help of dataset-specific statistical information. For SPARQL the cardinality estimation has to be applied on basic graph patterns and the graph structure defined by them. Thus, estimation techniques from relational databases, which often rely on the *attribute independence* and *uniform distribution* assumption, cannot be adopted as such. Instead, correlation in the RDF data has to be taken into account as well. In order to minimize the deviations between the estimated cardinality and the true results size detailed statistics are required. However, there may be restrictions regarding the availability of sophisticated statistical data, e. g. the VOID descriptions used by SPLENDID are usually limited with respect to the level of detail. Therefore, the cardinality estimation also has to cope with less accurate data source descriptions.

This section gives a detailed overview of the cardinality computation for SPARQL graph patterns. It starts with basic triple patterns and then extends it to more complex expression with conjunctive joins of triple patterns. Similar to the cost estimation of query plans the cardinality computation also employs a recursive algorithm. Specific problems regarding the correlations in RDF data will be discussed where appropriate.

5.4.1 Triple Pattern Cardinality Estimation

Triple patterns are the basic building blocks of SPARQL queries. They contain variables or constant values (i. e. bound variables) in subject, predicate, and object positions. The bound variables define restrictions for the RDF triples to be matched. Due to the similarity with SQL it is reasonable to adapt common cardinality estimation techniques from relational databases for SPARQL triple patterns but with slight modifications that suit the RDF data model.

Definition 5.1. *The cardinality of a triple pattern $\mathcal{P} = (s, p, o)$ with respect to a collection of datasets D from the Linked Data cloud is defined as*

$$card_D(\mathcal{P}) = \sum_{\forall\ d \in D} card_d(s, p, o) \tag{5.1}$$

Relational database often assume *independent attributes* in order to simplify the cardinality computation or if correlations are not known. An adaption for SPARQL triple patterns translates into following formula which considers the restrictions of the (bound) subject, predicate, and object variables with independent selectivity factors.

Definition 5.2. *The selectivity-based cardinality estimation for a triple pattern* (s, p, o) *with respect to a dataset* $d \in D$ *is the product of the dataset size with the specific selectivities of the subject, predicate, and object variables.*

$$card_d(s, p, o) = |d| \cdot sel_d(s) \cdot sel_d(p) \cdot sel_d(o) \tag{5.2}$$

where $|d|$ *is the dataset size, i. e. the number of all RDF triples in* d*, and the respective selectivity* sel_d *is a real number in the range* $[0..1]$.

The selectivity $sel_d(v)$ defines a factor by which a restriction of variable v reduces the number of matched RDF triples compared to the overall number of RDF triples in dataset $d \in D$. Unbound variables have a selectivity of 1. Hence, a triple pattern without bound variables matches all RDF triples in a dataset, i. e.

$$card_d(?s, ?p, ?o) = |d|. \tag{5.3}$$

A triple pattern with all variables bound can match at most one RDF triple which has the exact terms as defined in the triple pattern (or none if the dataset does not contain the specified RDF triple). For this special case the cardinality is set to 1.

$$card_d(\mathsf{s}, \mathsf{p}, \mathsf{o}) = 1. \tag{5.4}$$

The independent variable assumption may be sufficient for querying normalized data in relational database tables. However, it is no suitable at all for triple patterns in SPARQL queries. Due to the high correlation between subject, predicate, and object it is necessary to use a better cardinality estimation than the selectivity based formula 5.2. In fact, the statistics from the VOID descriptions can be used to obtain much better cardinality estimates for most variable combinations. In total, there exist eight different permutations for having bound and unbound variables in a triple pattern. Depending on the number and position of the bound variables different estimation approaches will be employed for calculating $card_d(s, p, o)$. In the following, the cardinality computation for triple patterns is distinguished between 1) triple patterns with schema-related bound variables, for which the *true cardinality* can be obtained from the VOID statistics, 2) triple patterns with single entity restriction, and 3) triple patterns with two correlated bound variables.

Triple Patterns with Schema Restriction

VOID descriptions may include statistics about the dataset schema in the so called property partition and class partition (cf. 5.3.4). This information about the number of occurrences of predicates and RDF types allows to determine the true cardinality for two kinds of triple patterns with specific schema-related variable restrictions, i. e. ($?s$, p, $?o$) and ($?s$, rdf:type, t), where $?s$, and $?o$ are unbound variables and p and t are constant RDF terms. The dataset-specific triple pattern cardinality is directly obtained by looking up the distinct counts for predicates and RDF types in all datasets $d \in D$.

$$\begin{aligned} card_d(?s, \mathsf{p}, ?o) &= \mathsf{VOID}card_{pred}(\mathsf{p}, d) \\ card_d(?s, \mathsf{rdf{:}type}, \mathsf{t}) &= \mathsf{VOID}card_{class}(\mathsf{t}, d) \end{aligned}$$

Triple Patterns with Entity Restriction

Determining the true cardinality for triple patterns with instance-specific bound variables, i. e. constants in subject or object position, cannot be done with the help of VOID statistics. Maintaining such information is prohibitively expensive due to the huge number of individual instances in the Linked Data cloud. Therefore, the cardinality is typically estimated based on the selectivity of the variable restriction.

The selectivity of an individual bound variable is often determined based on the uniform distribution assumption, i. e. each unique value has the same number of occurrences in the dataset. Hence, for SPARQL triple patterns the selectivity of a bound subject or object variable is computed as

$$sel_d(\mathsf{s}) = \frac{1}{|\{s \mid (s,p,o) \in d\}|}, \qquad sel_d(\mathsf{o}) = \frac{1}{|\{o \mid (s,p,o) \in d\}|} \tag{5.5}$$

where the denominator of the fractions represents the distinct number of subjects and objects, respectively. In VOID this kind of information is provided by void:distinctSubjects and void:distinctObjects statements. Of course, the *uniform distribution* assumption is imperfect, especially since there are typically very popular entities in a datasets and others which occur only once. Moreover there is no distinction between resources and literals. But since VOID statistics cannot provide detailed information about the potentially large number of entities in an RDF graph, this assumption is the best approximation which can be made. However, a future extension of VOID, e. g. with information about the average selectivity of resources and literals would be useful, though.

Cardinality Estimation for Dependent Variables

Triple patterns with two bound variables basically define restrictions for matching attributes of resources. From the three different combinations

$$(\mathsf{s}, \mathsf{p}, ?o), \qquad (?s, \mathsf{p}, \mathsf{o}), \qquad (\mathsf{s}, ?p, \mathsf{o})$$

the first defines a resource with an attribute type but the value is unknown, the second defines an attribute type and a value but the resource is unknown, and the third defines a resource with an attribute value but the type of the attribute is unknown.

The cardinality of these triple patterns can essentially be computed with formula 5.2 using separate selectivity estimates for the two bound variables. However, such a multiplication of selectivity values assumes *attribute independence* while combinations of attribute values in RDF are in general highly correlated. Thus, it is necessary to use additional statistical information in order to better estimate the selectivity of specific value combinations.

Two of the three combinations shown above have a bound predicate. Thus, the exact number of triple patterns for this predicate is known from the statistics. The additional bound subject or bound object reduces the result set accordingly to RDF triples with the respective value combination. This case is basically similar to triple patterns where only the subject or object is bound. The difference is that the base cardinality of the source data is not the overall number of RDF triples in the dataset but just the number of triples with the respective predicate. Hence, the co-occurrence cardinality is computed as

$$\begin{aligned} card_d(\mathsf{s}, \mathsf{p}, ?o) &= card_d(?s, \mathsf{p}, ?o) \cdot \frac{1}{|\{s \mid (s, \mathsf{p}, o) \in d\}|} \\ card_d(?s, \mathsf{p}, \mathsf{o}) &= card_d(?s, \mathsf{p}, ?o) \cdot \frac{1}{|\{o \mid (s, \mathsf{p}, o) \in d\}|}, \quad \mathsf{p} \neq \mathsf{rdf{:}type} \end{aligned} \tag{5.6}$$

Moreover, the number of distinct subjects and objects does not refer to the whole dataset as in equation 5.5 but also just to the subset of triples which contain predicate p. This information is provided in the predicate partitions of VOID statistics via predicate-specific `void:distinctSubject` and `void:distinctObject` values (cf. Fig. 5.4).

For the combination of a bound subject with a bound object, there is no additional information available. However, there is typically just one predicate which connects specific subjects and objects. Hence, the cardinality of such a combination is set to 1 by default.

$$card_d(\mathsf{s}, ?p, \mathsf{o}) = 1 \tag{5.7}$$

5.4.2 Cardinality of Basic Graph Patterns

A SPARQL basic graph pattern is a set of triple patterns (with optional filter expressions) which represents a conjunctive query. The triple pattern typically have shared variables, i. e. the join of two triple patterns represents an *Equi-Join*. Again, the cardinality estimation for joined triple patterns can be adapted from common join cardinality estimation techniques in relational databases [179].

Definition 5.3. *The cardinality of two joined triple patterns $\mathcal{P}_1$ and $\mathcal{P}_2$ is defined as the cardinality of the Cartesian product of both triple pattern multiplied with the respective join selectivity.*

$$card_d(\mathcal{P}_1 \bowtie \mathcal{P}_2) \;=\; card_d(\mathcal{P}_1 \times \mathcal{P}_2) \cdot sel_d(\mathcal{P}_1 \bowtie \mathcal{P}_2) \tag{5.8}$$

where $card_d(\mathcal{P}_1 \times \mathcal{P}_2) = card_d(s_1, p_1, o_1) \cdot card_d(s_2, p_2, o_2)$.

The cardinality of the Cartesian product defines an upper bound for the result size. However, due to the restrictions of the shared variables the actual result cardinality will typically be much smaller. Therefore, the join selectivity represents an estimation of the fraction of results which are actually matched by the join condition. An exception are triple patterns without shared variables. Then the join selectivity will be 1.

In relational databases the selectivity for Equi-Joins is computed by estimating the overlap between the values of the respective join attributes (typically relying on the *attribute independence*) assumption. Having two relations R and S with join attributes $R.a$ and $R.b$, the join selectivity can be expressed as the maximum of the individual selectivities of the join attributes.

$$sel(R \bowtie_{R.a=S.b} B) = max(sel(R.a), sel(S.b)) \tag{5.9}$$

However, RDF data is highly correlated and joins in SPARQL are basically *self-joins* on all RDF triples in a dataset. Hence, the join selectivity depends on the actual position of the shared variable in the joined triple patterns, i. e. subject-subject joins are considered more selective than subject-object joins or object-object joins. Moreover, joins involving predicates are usually most selective. These correlations can hardly be captured with the join selectivity estimation of equation 5.9. Instead, the graph structure defined by the basic graph patterns has to be considered for a reliable estimation the appropriate join selectivity. Following, the two most prominent join patterns are distinguished, i. e. star-shaped graph patterns, which are used to match resources with certain attributes, and path-shaped graph patterns, which match connections between entities.

Star-Join Cardinality

Star shaped query patterns are very common in SPARQL queries [184]. A join variable which occurs in subject position of multiple triple patterns selects all resources with the matching set of attributes. Example 5.4 shows two SPARQL queries with different basic graph patterns which are joined via a shared subject variable $?s$. In the first query all predicates and all objects are bound while in the second query there are also unbound predicates and unbound objects. This distinction is significant for computing the cardinality of the star-join.

Example 5.4 (Star-Join). Two queries with different basic graph patterns: one with bound predicates and objects (left) and the other with unbound predicate and unbound object (right).

```
1 SELECT *
2 WHERE {
3   ?s a foaf:Person.
4   ?s :lives_in "Berlin".
5   ?s foaf:knows :John .
6 }
```

```
1 SELECT *
2 WHERE {
3   ?s a foaf:Person.
4   ?s ?relation "Berlin".
5   ?s foaf:knows ?someone .
6 }
```

A triple pattern with a bound predicate and a bound object imposes a greater restriction on the result size because only value bindings for one variable will be returned. In the example there is just the variable $?s$ in the first query but three variables in the second query, namely $?s$, $?relation$, and $?someone$. Consequently, there are more possible combinations for variable bindings in the latter case. In fact, the result size for the first case is limited by the triple pattern with the smallest cardinality. For example, assuming that there are n entities which match $<?s$:lives_in "Berlin"$>$ and m entities which match $<?s$:has_name "John"$>$, then there cannot be more than $min(n, m)$ entities which match both triple patterns. In contrast, for the second case the result size can be increased by adding variable bindings for $?someone$ because each entity which matches $?s$ can have multiple foaf:knows relations. Hence, the cardinality estimation for star-joins employs a distinction between the different triple pattern types.

Definition 5.5 (Star-Join Cardinality). *Let $\mathcal{P}_1, \ldots, \mathcal{P}_n$ be a set of triple patterns with a shared subject variable. Further, let $\mathcal{P}''$ be a triple pattern with two bound variables and let $\mathcal{P}'$ be a triple pattern with one bound variable such that $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\} = \{\mathcal{P}''_1, \ldots, \mathcal{P}''_k\} \cup \{\mathcal{P}'_1, \ldots, \mathcal{P}'_l\} \wedge k + l = n$. Then, the star-join cardinality is defined as*

$$card^*_D(\mathcal{P}_1, \ldots, \mathcal{P}_n) = \sum_{\forall\, d \in D} card^*_d(\mathcal{P}_1, \ldots, \mathcal{P}_n) \tag{5.10}$$

with

$$card^*_d(\mathcal{P}_1, \ldots, \mathcal{P}_n) = min(card_d(\mathcal{P}''_i)) \cdot \prod card^s_d(\mathcal{P}'_j) \tag{5.11}$$

where $card^s_d(\mathcal{P}'_j)$ is the cardinality of $\mathcal{P}'_j$ in combination with a bound subject variable s.

This formula cannot be applied if individual statements with the same subject are mapped to different data sources. In that case, the Star-Join cardinality is computed separately for each subset and aggregated using the formula for Path-Joins (Def. 5.6).

The multiplication with $card_d^s$ allows for taking into account the increasing result size when triple patterns with unbound predicate or unbound object are involved. Its computation is done using the equations for dependent variables in Sec. 5.4.1. For example, if 500 RDF triples with predicate foaf:knows have 50 distinct subjects, then the cardinality for the respective triple pattern in combination with a bound subject is 10, because on average each subject will have 10 foaf:knows relations.

Due to missing information about the correlation of triple patterns in a star-join, e. g. if predicates co-occur for the same subjects, it is hardly possible to further improve the cardinality estimation. However, the estimate already gives a good and reliable upper bound for the star-join result size. Finally, analogous to the subject-based star-join, the same computation can be used to estimate the cardinality of an object-based star-join.

Path-Join Cardinality

A sequence of connected SPARQL triple patterns, i. e. with shared variables occurring in subject position of one triple pattern and in object position of another triple pattern, is called a path-join. The cardinality estimation for path-joins extends equation (5.8) such that sequences of triple patterns across multiple data sources are covered.

Definition 5.6 (Path-Join Cardinality). *For a sequence of triple patterns $\mathcal{P}_1, \ldots, \mathcal{P}_n$, which represent a path-join over a set of data sources D, the path-join cardinality is defined as*

$$card_D^{\times}(\mathcal{P}_1, ..., \mathcal{P}_n) = \prod_{1<i<n} card_D(\mathcal{P}_i) \;\cdot\; sel_D(\mathcal{P}_1, \mathcal{P}_2) \cdot \ldots \cdot sel_D(\mathcal{P}_{n-1}, \mathcal{P}_n) \tag{5.12}$$

The computation of $card_D(\mathcal{P}_i)$ follows equation (5.1) while the join selectivity considers multiple paths across different data sources by averaging over the individual selectivities for each data source combination.

$$sel_D(\mathcal{P}_i, \mathcal{P}_{i+1}) = avg(sel_{d,d'}(\mathcal{P}_i, \mathcal{P}_{i+1})) \qquad \forall\, d, d' \in D: \; card_d(\mathcal{P}_i) > 0 \wedge card_{d'}(\mathcal{P}_{i+1}) > 0 \tag{5.13}$$

Since VOID descriptions do not include statistics about join paths, especially not for joins across data sources, the selectivity computation relies on the distinct number of subjects and objects, assuming a uniform distribution.

$$sel_{d,d'}(\mathcal{P}_i, \mathcal{P}_{i+1}) = max(sel_d(o_i), sel_{d'}(s_{i+1})) \tag{5.14}$$

5.5 Evaluation

In order to assess the effectiveness of the SPLENDID query optimization, an evaluation with a federation of actual Linked Data sources had to be done.

The main objective was to verify that distributed SPARQL queries can be processed efficiently with SPLENDID by choosing optimal query execution plans. Therefore, measuring the actual query processing time (including source selection, query optimization, and query execution) is the most comprehensive indicator for the overall performance. An isolated evaluation of the cardinality estimation has not been conducted because initial investigations have shown that although the magnitude of the cardinality estimation error grows quickly with the number of joins it has little effect on the final join order in a query plan. Hence, the evaluation concentrated on the query execution time

The evaluation methodology employs a variation of query optimizer settings in order to investigate how the performance of the query evaluation is influenced by choosing different (physical) query execution plans. Therefore, the use of hash join and bind join in SPLENDID query plans is evaluated separately. Moreover, the effects of combining owl:sameAs patterns in sub queries is evaluated. This combination can significantly influence the query processing performance but it should only be applied if no third party links are involved (cf. Sec. 4.3.3). In the second part the overall effectiveness of the SPLENDID query optimization is evaluated through a comparison with state-of-the-art federation implementations, i. e. DARQ [188], FedX [203], and Sesame AliBaba[2]. The main differences between the systems (cf. Table 5.1) are the use of heuristics (FedX, AliBaba) versus cost-based query optimization (SPLENDID, DARQ) and the choice of physical join implementations.

Table 5.1. Characteristics of the compared state-of-the-art federation systems. Physical Join implementations are Hash Join (HJ) and Nested-Loop Join (NLJ) for local join computation as well as Bind-Join (BJ) [93] and Controlled Bound Join (CBJ) [203] for reducing the data transfer cost.

	SPLENDID	DARQ	FedX	AliBaba
query plan structure	bushy	left-deep	left-deep	left-deep
optimization algorithm	dyn. prog.	it. dyn. prog.	heuristics	heuristics
supporting statistics	yes (VOID)	yes	no	no
join implementations	HJ, CBJ	NLJ, BJ	CBJ	NLJ, BJ

A comparison of different federation systems typically considers each implementation as a black box and does not evaluate the different components independently, i. e. query optimization and query execution. However, preliminary evaluation results indicated that FedX has generally a superior query execution due to the optimized bind join implementation which significantly

[2] http://www.openrdf.org/doc/alibaba/2.0-beta9/alibaba-sail-federation/index.html

reduces the network communication cost. In order to obtain comparable evaluation results considering only the effectiveness of the query optimization, it is necessary to perform the evaluation without side effects from the source selection and query execution. Unfortunately, the implementations of the compared federation systems do not easily allow for an isolated evaluation. In fact, some components are tightly coupled, e. g. the query execution needs to support the physical join implementations used in the query optimizations and vice versa. Therefore, the federation implementations of DARQ and AliBaba were evaluated as is. Only for SPLENDID and FedX it was possible to handle the query optimization separately and use the same source selection and query execution code, i. e. the source selection was done in both cases with ASK queries for each triple pattern and for the query execution SPLENDID employs the Blocked Bind Join implementation of FedX as well.

5.5.1 Setup

The comparison of different optimization strategies is best done with a benchmark. However, the number of suitable benchmarks for distributed SPARQL query processing is very limited. In fact, FedBench [198] is the only benchmark suite which explicitly focuses on this scenario and has already been used by a number of researcher for this purpose. Other research efforts in this direction [89, 164] mainly investigate improvements for distributed SPARQL benchmarks but do not provide a complete benchmark suite. The FedBench evaluation infrastructure[3] provides means for measuring the average query execution times of different benchmark queries. It incorporates snapshots of ten Linked Data sources, separated in life science and cross domain datasets, and 14 queries with different characteristics which cover these datasets. Details of the benchmark queries are shown in Table 5.2. FedBench prefers dataset snapshots over live requests on SPARQL endpoints because the latency and bandwidth can differ significantly. It would be hardly possible to obtain reliable results for a detailed performance comparison.

Table 5.2. Number of covered data sources and result sizes for FedBench's cross domain (CD) and life science (LS) queries

	CD1	CD2	CD3	CD4	CD5	CD6	CD7	LS1	LS2	LS3	LS4	LS5	LS6	LS7
sources	2	2	5	5	5	4	5	2	4	2	2	3	3	3
results	90	1	2	1	2	11	1	1159	333	9054	3	393	28	144

The SPLENDID evaluation setup comprised five 64bit Intel® Xeon® CPU 3.60GHz server instances running Sesame 2.4.2. Each of the servers provided

[3] http://code.google.com/p/fbench/

the SPARQL endpoint for one life science and for one cross domain dataset. The SPLENDID federator instance was executed on a separate server instance with 64bit Intel® Xeon® CPU 3.60GHz and a 100Mbit network connection. Each FedBench query was evaluated ten times in order to measure the average query execution time. The first run was excluded such that cold caches could not influence the results. The query execution time was measured as the time required for retrieving the complete result set. In order to prevent extremely long running queries a timeout of two minutes was set.

5.5.2 Results

The first evaluation compares the effects on the query execution performance for varying physical join implementations in the query plans, i. e. the SPLENDID optimizer used either hash join or bind join exclusively in the generated query plan. Additionally, the effect of grouping owl:sameAs patterns in sub queries was investigated. The second part shows the results for comparing the performance with different state-of-the-art federation implementations.

Comparison of Different Join Implementations

SPLENDID was run with different settings for applying the two physical join implementations in the query plans. Figure 5.5 shows the results. The first interesting observation can be made regarding the query timeout for CD1, CD6, CD7, LS, and LS3 with hash joins. In contrast, all queries with bind joins finished within the time limit. The bind join is also superior to the hash join in almost all queries. However, the life science queries LS1, LS5, and LS7 are executed fastest with the hash join implementation. The reason for these observations are the different sizes of intermediate result sets. The bind join has a clear advantage when the amount of data which has to be transmitted over the network is reduced. The hash join is better when only small intermediate results have to be retrieved.

Figure 5.6 shows the query execution times for query plans where both bind join and hash join can be applied in a query plan. All cross domain queries achieve the best query execution time compared to Fig. 5.5. However, for the life science queries it also happens that a sub optimal query plan is chosen, which can be attributed to estimation errors in the cardinality and cost computation for the query plans.

The grouping of owl:sameAs patterns in the cross domain queries leads generally to faster query execution times. The highest impact can be seen for the cross domain queries. The life science queries do not contain owl:sameAs patterns except for LS3. However, such a grouping of owl:sameAs patterns only returns complete results if the owl:sameAs triple statements are located in the same data source as the subject (which is the case for FedBench).

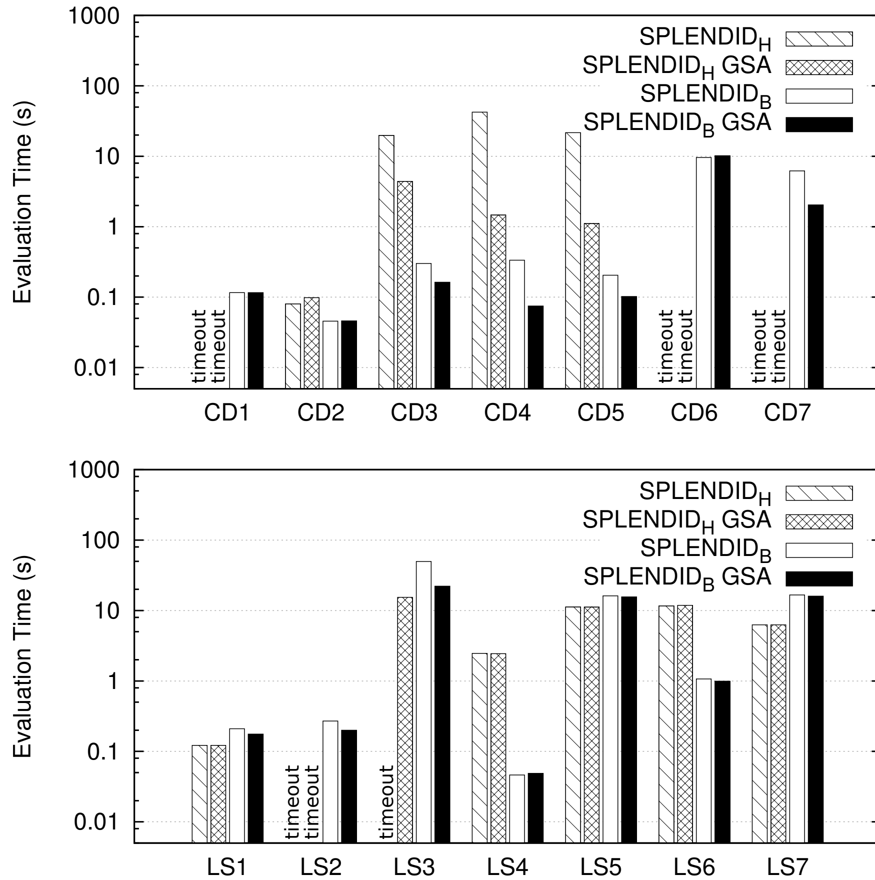


Fig. 5.5. SPLENDID query execution times for cross domain (CD) and life science (LS) queries. The optimization employs either bind join (B) or hash join (H). The grouping of owl:sameAs patterns (GSA) further reduces the query execution time.

Comparison with State-of-the-Art Federation Systems

The results of the comparison of the query execution times for SPLENDID, FedX, DARQ and AliBaba are shown in Fig. 5.7. DARQ and AliBaba are not able to process all FedBench queries. AliBaba generates malformed sub queries for CD3, CD5, LS6, and LS7. DARQ cannot evaluate CD1 and LS2 due to unbound predicates and CD3 and CD5 fail to complete because DARQ opens too many connections to the GeoNames endpoint. DARQ and AliBaba queries labeled with *timeout* take longer than the limit of two minutes. SPLENDID and FedX can return results for all queries within this time limit.

FedX and SPLENDID achieve for all queries the best query execution times when compared with DARQ and AliBaba. FedX is usually a bit faster than SPLENDID, except for queries CD3 and CD4. An investigation of theses differences revealed interesting insights. The FedX optimization heuristics are faster than the cost-based query optimization in SPLENDID. But usually both FedX and SPLENDID generate the same query execution plans with the same join order. In some cases SPLENDID uses hash joins instead of bind joins and produces a different join order for two queries. This lead to a sub optimal query plan for queries CD6, CD7, LS5, LS6, LS7. The reason are erroneous estimates

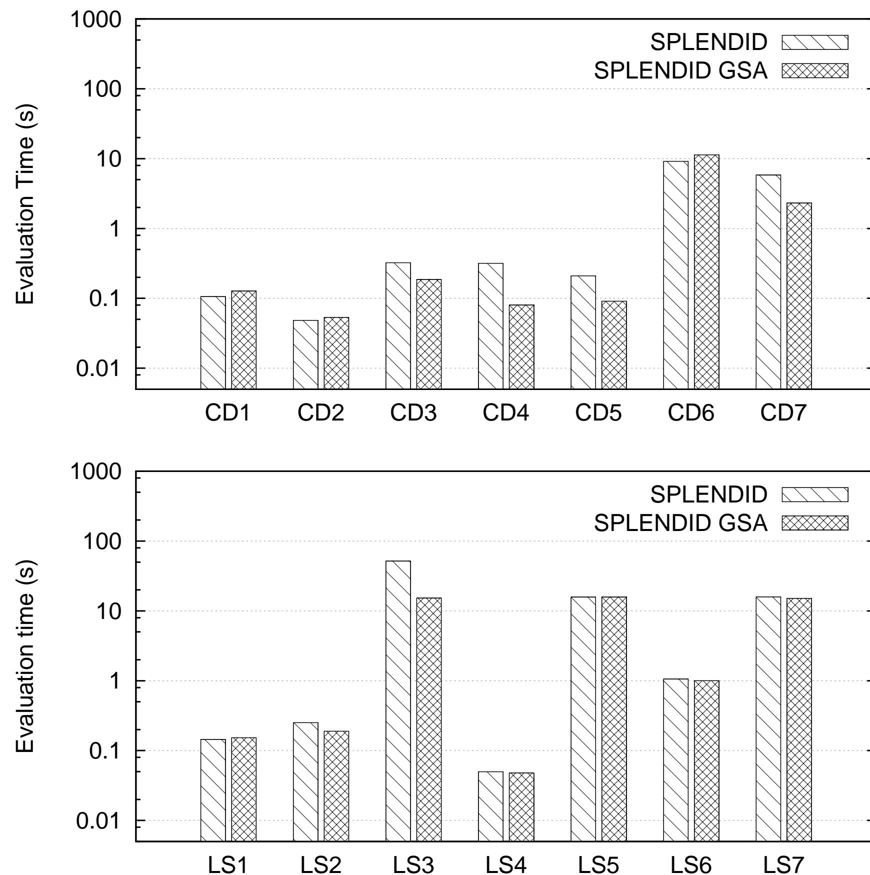


Fig. 5.6. SPLENDID query execution times for cross domain (CD) and life science (LS) queries using a combination of bind join and hash join. The effect of grouping of owl:sameAs patterns (GSA) is shown as well.

for the result cardinality and the query execution cost. Additionally, the hash join is hardly optimized and relies on accurate estimates for the cardinalities of the input data. However, the results of SPLENDID for queries CD3 and CD4 supports the hypothesis that better query performance can be achieved through cost-based query optimization and different join operator implementations. An independent investigation by Rakhmawati and Hausenblas [189] confirm the good performance of SPLENDID for different data distributions and partitions.

5.6 Summary

This chapter started with a survey of relevant research in the area of distributed and federated databases. It has been shown that there are a lot of useful query optimization techniques which can be adopted for distributed SPARQL query processing on federated Linked Data source. However, there are also significant differences concerning the RDF data structure and limitations of the SPARQL syntax and protocol.

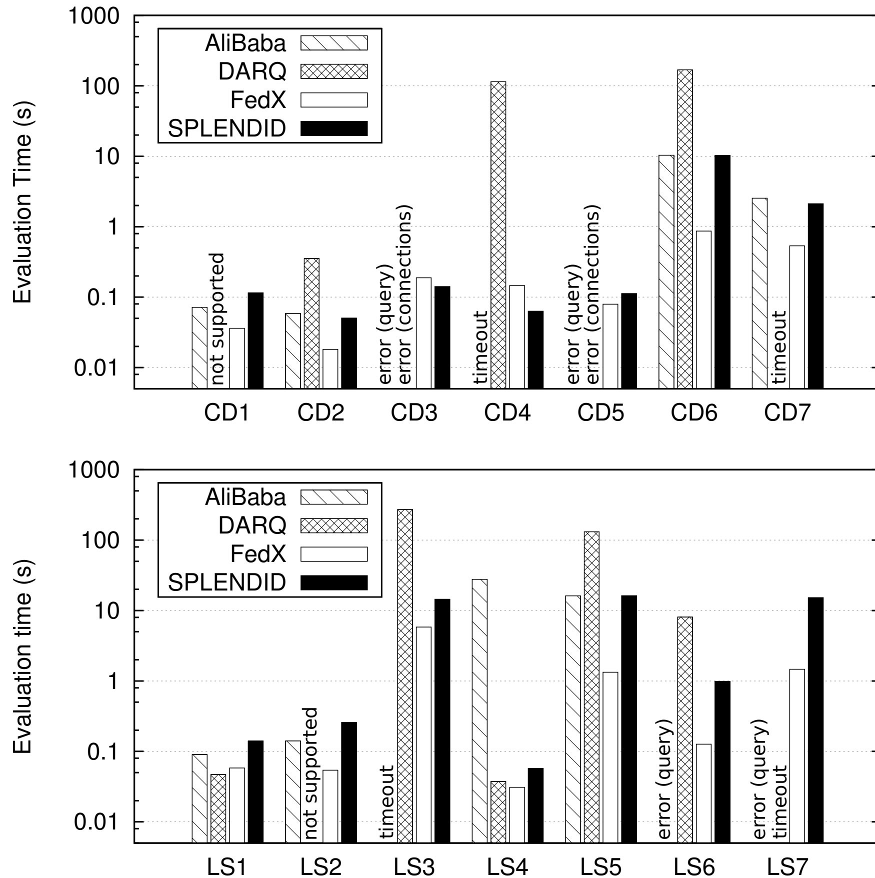


Fig. 5.7. Comparing the query evaluation time for state-of-the-art SPARQL endpoint federation approaches, i. e. Sesame AliBaba, DARQ, FedX, and SPLENDID, using the FedBench cross domain (CD) and life science (LS) queries.

The main contributions to this field of research are the newly developed SPLENDID approach for implementing effective distributed query processing on Linked Data employing cost-based query optimization which used statistical data expressed in VOID descriptions. Therefore, a cost model based on data transfer cost and suitable cardinality estimation techniques capturing the correlation of RDF data were derived.

The final evaluation has shown that efficient physical join implementations, like hash join and bind join, have a significant influence on the query execution performance. With respect to finding the best query execution plan, it turns out that heuristics can provide quite good results for queries with common basic graph patterns. The optimization of more complex queries is certainly more challenging and SPLENDID shows a slightly better performance in some of these cases. However, the complexity of the FedBench [198] benchmark queries, e. g. with respect to the number of joins and the number of involved data sources, is limited. Therefore, chapter 7 investigates the benchmarking of Linked Data federation systems in more depth.

6

PINTS: Maintaining Distributed Statistics for Peer-to-Peer Information Retrieval

RDF is a very flexible data format which can be used to represent information from various domains. Depending on the application scenario, large scale RDF data management can be implemented based on different infrastructure paradigms (cf. Chap. 3). While the previous chapters focused on federated SPARQL query processing over distributed, linked RDF data sources, this chapter deals with the maintenance of distributed statistics for graph-structured data in Peer-to-Peer networks. Most of the research on RDF data storage in Peer-to-Peer networks (cf. Sec. 3.6) investigates optimal data distribution and sophisticated query processing strategies. But a ranked retrieval of peer-specific search results is often not covered. In general, the search on RDF graphs differs significantly from the classic document-based information retrieval, i. e. SPARQL queries either return tuples with variable bindings, individual RDF triples, or specific RDF sub graphs. Therefore, a relevance ordering on the results essentially requires specific ranking strategies, e. g. [113, 81, 158]. However, these algorithms do not fit well to the data storage in structured Peer-to-Peer networks. Gathering all required data at one peer (i. e. the query initiator) and then obtaining a list of globally ranked results based on locally computed relevance scores is typically prohibitively expensive. A restriction of the application scope to a specific domain can help to reduce the complexity because the underlying data model will be limited to a well known schema with specific entity types.

Typical scenarios for the application of Peer-to-Peer systems are large user communities which exchange information or share specific resources, e. g. social networks and file sharing communities are the classic use cases. From an infrastructure perspective basically any social network can be managed in a distributed fashion on top of a Peer-to-Peer network. An example for such a

system is Tagster [86] which provides support for decentralized collaborative tagging using a tagging ontology [174]. The tagging ontology defines an RDF model for the relation between users, tags, and resources. Search requests on such a collaborative tagging network are user-centric, resource-centric, or tag-centric. Due to the similarity to classic keyword-based information retrieval it is possible to adapt well known concepts, like the *vector space model* and the *tf-idf metric* [17], and apply relevance ranking by computing the similarity between tagging-based feature vectors. However, the computation of *tf-idf* in a Peer-to-Peer system is challenging because some statistics, e. g. the co-occurrence count for related users, tags, and resources, need to be collected from different peers in the network. Therefore, it is necessary to implement an efficient meta-data management strategy which can basically aggregate and maintain important statistics with minimal network communication overhead. In order to solve this problem, the contribution of this chapter is twofold. First, a formal model for characterizing item sets in a collaborative tagging system is defined and used as the basis for adapting the *tf-idf* metrics for the underlying tripartite network structure. Second, PINTS [85], an efficient solution for managing distributed statistics, is presented. It employs a prediction-based update strategy for global statistics which significantly reduces the network communication cost and, at the same time, maintains a high accuracy of the individual feature vectors.

The chapter is structured as follows. Section 6.1 gives an overview of the collaborative tagging model along with a description on how the traditional *tf-idf* approach can be adapted to it. A description of the challenges for distributed collaborative tagging is given in Sec. 6.2. Then, in Sec. 6.3, the PINTS approach for efficient updates of statistics in the distributed scenario is presented. Finally, a discussion of evaluation results based on real tagging datasets follows in Sec. 6.4.

6.1 A Data Model for Collaborative Tagging Systems

Collaborative tagging systems have gained some popularity in the last decade. Tagging is a well suited paradigm for data organization and allows for easy categorization of information objects by assigning simple user-chosen keywords (i.e. tags). People tag resources for two different reasons: for personal data organization and to share interesting information with others. This gave rise to services like Flickr[1], Delicious[2], YouTube[3], LastFM[4], Bibsonomy[5] and the like which allow for easy organization and sharing of photos, bookmarks, videos, music, and publications over the Internet. The main benefit of such

[1] http://flickr.com
[2] http://delicious.com
[3] http://youtube.com
[4] http://last.fm
[5] http://bibsonomy.org

sites is the collaborative categorization of a huge amount of data and the ability to browse in a very easy way through all that data to find new interesting things because other people have been tagging them with similar tags [84].

Tagging systems are not bound to a fixed classification hierarchy. A user simply assigns any number of arbitrary, short labels (*tags*) to a resource in order to categorize it. Figure 6.1 illustrates the relations between users, tags, and resources as defined through different tag assignments. Resource retrieval is then performed on the basis of the associated tags and users can simply browse along the *tag-resource-user relations*. The emerging network, which has similar characteristics as complex systems [84, 50, 96], is also called a *folksonomy* (derived from *folk+taxonomy*[6]).

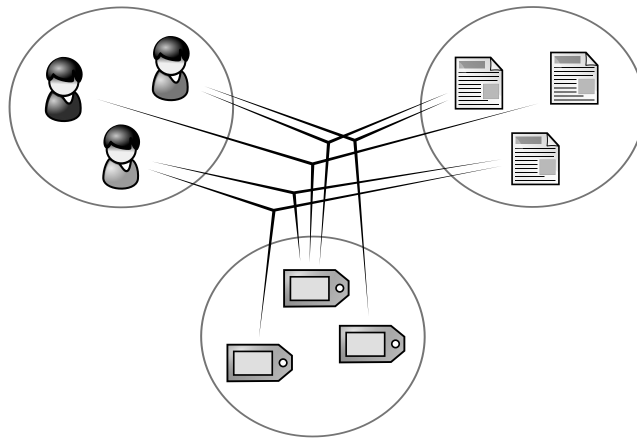


Fig. 6.1. Tag assignment relations between *users*, *tags*, and *resources*.

6.1.1 Tripartite Networks

Folksonomies have the structure of a *tripartite network* [138] (or *hypergraph*) with *ternary relations* (i. e. tag assignments) between *users*, *resources* (e. g. images, media files), and associated *tags* (i. e. arbitrary text labels).

Definition 6.1. *Folksonomy. Let U be a set of users, T be a set of tags, and R be a set of resources. Then a set of tag assignments is $Y \subseteq U \times T \times R$.*

The elements in U, T, and R can be mutually characterized based on these relations. For example, a user's interests and the topic of a resource can be described by the associated tags. Analyzing such characteristics [201] helps to find interesting resources in a folksonomy, identify specific interest groups, give personalized recommendations, or provide tag suggestions for annotating new resources. Computations on folksonomy data often rely on common network analysis approaches which require that tripartite tag assignments are projected to a bipartite or unipartite network [84], e. g. by counting the frequency of occurrences or the correlation between elements.

[6] http://vanderwal.net/folksonomy.html

6.1.2 Generic Tag Clouds

The notion of “tag clouds” is commonly used to visualize tag-centered folksonomy characteristics. Figure 6.2 shows an example tag cloud centered around the topic Web 2.0. It uses different font sizes to differentiate between important tags (large font) and less important tags (small font). Additionally, different colors can be used or the tag layout and sort order may be varied.

Fig. 6.2. Example of a tag cloud centered around the topic “Web 2.0”[7]

However, various forms of relations between users, tags, and resources are interesting from an information retrieval perspective. Therefore, the concept of a *generic* “tag cloud” is introduced to characterize elements from U, R, and even T in a flexible way.

Definition 6.2. *A generic tag cloud is defined on a context-dependent subset* $Y^* \subseteq Y$ *of all tag assignments using a tag-rank function* $f(t)$ *that computes a score (weight) for each* $t \in T^*$ *from* Y^**.*

$$\mathcal{T} := (Y^*, f) \quad with \quad f(t) : Y^* \to \mathbb{R} \tag{6.1}$$

Using Def. 6.2 individual users and resources can be characterized based on the collection of tags from the respective subset of relations in Y^*. Thus, it is possible to differentiate between *user-centric* tag clouds $\mathcal{T}_u$ (i. e. all tags assigned by a user $u \in U$) and *resource-centric* tag clouds $\mathcal{T}_r$ (i. e. all tags associated with a specific resource $r \in R$):

$$\mathcal{T}_u := (Y_u, f_u), \quad Y_u \subseteq \{u\} \times T \times R \tag{6.2}$$

$$\mathcal{T}_r := (Y_r, f_r), \quad Y_r \subseteq U \times T \times \{r\} \tag{6.3}$$

Moreover, *community-centric* tag clouds can be defined which summarize all tags of a group of users $U^* \subseteq U$ or for a collection of resources $R^* \subseteq R$:

$$\mathcal{T}_{U^*} := (Y_{U^*}, f_{U^*}), \quad Y_{U^*} \subseteq U^* \times T \times R \tag{6.4}$$

$$\mathcal{T}_{R^*} := (Y_{R^*}, f_{R^*}), \quad Y_{R^*} \subseteq U \times T \times R^* \tag{6.5}$$

The combination of a group of users $U^* \subseteq U$ and a collection of resources $R^* \subseteq R$ leads to a tag cloud centered around a community of user and resources:

$$\mathcal{T}_{U^*R^*} := (Y_{U^*R^*}, f_{U^*R^*}), \quad Y_{U^*R^*} \subseteq U^* \times T \times R^* \tag{6.6}$$

The conceptual model behind these tag cloud definitions is rather generic and can be easily adapted to focus on users and resources. Although "user clouds" or "resource clouds" are rather uncommon, a generic tag cloud is essentially a collection of labels with an associated weight which is interpreted as the relevance of the labeled item in a specific context. Therefore, it is convenient to stick to the popular tag cloud notion, even if the concept is applied in a broader context where it represents a collection of labeled items from U, T, or R, respectively.

6.1.3 Feature Vectors for Folksonomies

Item-based searches in folksonomies, such as finding all resources or users associated with a given tag, can be realized by employing common information retrieval approaches. However, an adaptation with respect to the characteristics of tag clouds and the tripartite tagging relations is necessary. The classic information retrieval [17, 152] usually relies on the *Vector Space Model*, which employs feature vectors to characterize documents and to determine their relevance with respect to the search terms.

The Vector Space Model

A feature vector $v = (w_1, w_2, \ldots, w_n)$ contains weights for the terms that occur in the document or query, respectively. Relevance ranking of documents is done by computing the similarity between the query terms and the features of the documents [17]. The difference between two feature vectors v_1 and v_2 is often calculated using the *cosine similarity measure*:

$$sim(v_1, v_2) = \frac{v_1 \cdot v_2}{||v_1|| \cdot ||v_2||} \tag{6.7}$$

This measure basically determines the angle between two vectors in the vector space (dot product normalized by the vector length $||v||$).

[7] http://wikipedia.org/w/index.php?title=File:Web_2.0_Map.svg, CC BY-SA 2.5

Term weights in a feature vector can be influenced by long documents and frequent terms. Therefore, the *tf-idf* weighting function takes into account the whole document collection and computes normalized weights for all terms based on *term frequency (tf)* and *inverse document frequency (idf).*

Term Frequency captures the number of occurrences of term t in document d. This can be the raw term frequency $freq(t,d)$ or (in order to accommodate for longer documents) a normalized frequency, e. g. raw term frequency divided by the maximum raw frequency of any term w in d.

$$tf(t,d) = \frac{freq(t,d)}{max(freq(w,d) : w \in d)} \tag{6.8}$$

Inverse Document Frequency takes into account the popularity of term t in the whole document collection D. It is the ratio of the overall number of documents divided by the number of documents the term occurs in.

$$idf(t,D) = \log \frac{|D|}{|\{d \in D : t \in d\}|} \tag{6.9}$$

The normalized importance of term t within document d relative to the document collection D is then computed as $tfidf(t,d,D) = tf(t,d) \cdot idf(t,D)$.

TF-IDF for Tag Clouds

The adaptation of *tf-idf* for tag clouds requires a modification of the statistics calculations in order to work for the tripartite relation of tags, users, and resources instead of the bipartite relation of terms and documents. Therefore, this section defines a generic weighting model which is employed to compute the weight for a user, tag, or resource based on the other two related sets.

Definition 6.3. *Let I be an item set, i. e. U, T, or R. Then, the domain of I is a tuple which contains the respective other two item sets J and K*

$$dom(I) = (J,K) \quad \textit{with } I,J,K \in \{U,T,R\} \textit{ and } I \neq J \neq K \tag{6.10}$$

For example, $dom(U) = (T,R)$, $dom(T) = (U,R)$, and $dom(R) = (U,T)$.

Users and resources can be characterized by their associated tags. In turn, the relevance of a tag is usually computed based on its occurrence in the set of users or in the document corpus, respectively. However, the *tf-idf* computation on the tripartite tagging relations allows for aggregating the information for resources and users in a combined tag weight. Following is a general formalization for *tf-idf* on arbitrary item sets which combines the weight computation of both item sets from the respective domain in one formula. Therefore, the notions of *item-to-item frequency* (if) and *inverse item frequency* (iif) are introduced, which return a vector with two weight entries, one for each dimension of the domain of an item set.

Definition 6.4. *For a context dependent subset of tag assignments Y^*, the item-to-item frequency (if) is defined as the number of occurrences of item $i \in I$ in a relation with distinct items from dom(I)*

$$if_{Y^*}(i) = (|J_i^*|, |K_i^*|) \tag{6.11}$$

$$\begin{aligned} \textit{with} \;\; J_i^* &= \{j \in J \mid \exists x \in K : (i, j, x) \in Y^*\} \\ K_i^* &= \{k \in K \mid \exists x \in J : (i, x, k) \in Y^*\}, \end{aligned}$$

e. g. the number of different users and resources associated with a tag.

The inverse item frequency (iif) represents the popularity of an item $i \in I$ in the whole set of tag assignments Y. It is defined on the item sets in dom(I) as the ratio between the respective item set cardinality and the subset of items which occur in a relation with item i.

$$iif_Y(i) = \left(\log \frac{|J|}{|J_i|}, \log \frac{|K|}{|K_i|}\right) \tag{6.12}$$

$$\begin{aligned} \textit{with} \;\; J_i &= \{j \in J \mid \exists x \in K : (i, j, x) \in Y\} \\ K_i &= \{k \in K \mid \exists x \in J : (i, x, k) \in Y\} \end{aligned}$$

Combining $if(i)$ and $iif(I)$ via the dot product (inner product) of the respective vector representation yields the value of the i-th feature vector element.

$$ifiif(i) = if(i) \cdot iif(i)^T. \tag{6.13}$$

In order to allow for a more flexible tuning of the influence of the item sets in dom(i), two arbitrary weighting parameters w_1, w_2 with $w_1 + w_2 = 1$ are introduced, which are multiplied as elements of a diagonal matrix W.

$$ifiif(i) = if(i) \cdot W \cdot iif(i)^T \qquad \textit{with} \quad W = \begin{pmatrix} w_1 & 0 \\ 0 & w_2 \end{pmatrix} \tag{6.14}$$

Following example illustrates the computation of user-specific feature vectors based on a small set of tag assignments Y with three users, three tags and three resources. Figure 6.3 visualizes the respective bipartite graph representation centered around the tags.

The construction of the tag-centric feature vector of Alice employs the subset Y_{Alice} for computing the *if* values of each tag (cf. Eq. 6.11) and all tag assignments in Y to determine the corresponding *iif* values (cf. Eq. 6.12).

$$v_{Alice} = \begin{pmatrix} if_{Y_{Alice}}(holiday) \cdot iif_Y(holiday) \\ if_{Y_{Alice}}(mountain) \cdot iif_Y(mountain) \\ if_{Y_{Alice}}(sea) \cdot iif_Y(sea) \end{pmatrix}$$

$$= \begin{pmatrix} (1,1) \cdot (\log(\frac{3}{1}), \log(\frac{3}{1}))^T \\ (1,1) \cdot (\log(\frac{3}{2}), \log(\frac{3}{2}))^T \\ (1,1) \cdot (\log(\frac{3}{2}), \log(\frac{3}{1}))^T \end{pmatrix} = \begin{pmatrix} 2 \cdot \log(3) \\ 2 \cdot \log(\frac{3}{2}) \\ \log(\frac{3}{2}) + \log(3) \end{pmatrix} \simeq \begin{pmatrix} 3.2 \\ 1.2 \\ 2.2 \end{pmatrix}$$

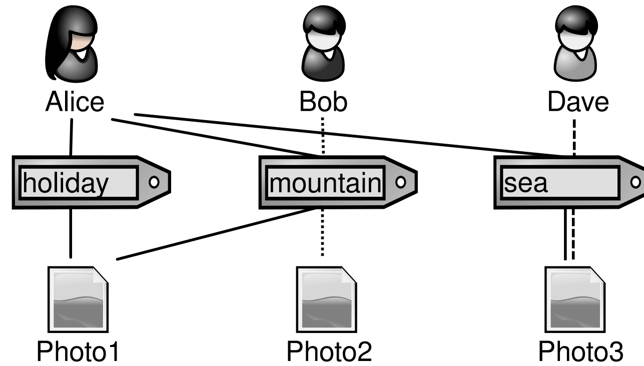


Fig. 6.3. Tag-centered visualization of five tag assignments by three users

A weight in the resulting feature vector is higher if the tag is more discriminative than another tag, i. e. if it is less popular and connected with fewer users or resources. For example, the tag "mountain" is shared by two users. Hence, it is less specific than the tag "holiday" which is only used by Alice. The feature vectors of Bob and Dave (using Y_{Bob} and Y_{Dave}, respectively) are then computed accordingly.

$$v_{Bob} = \begin{pmatrix} (0,0)\cdot(\log(\frac{3}{1}),\log(\frac{3}{1}))^T \\ (1,1)\cdot(\log(\frac{3}{2}),\log(\frac{3}{2}))^T \\ (0,0)\cdot(\log(\frac{3}{2}),\log(\frac{3}{1}))^T \end{pmatrix} = \begin{pmatrix} 0 \\ 2\cdot\log(\frac{3}{2}) \\ 0 \end{pmatrix} \simeq \begin{pmatrix} 0 \\ 1.2 \\ 0 \end{pmatrix}$$

$$v_{Dave} = \begin{pmatrix} (0,0)\cdot(\log(\frac{3}{1}),\log(\frac{3}{1}))^T \\ (0,0)\cdot(\log(\frac{3}{2}),\log(\frac{3}{2}))^T \\ (1,1)\cdot(\log(\frac{3}{2}),\log(\frac{3}{1}))^T \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \log(\frac{3}{2})+\log(3) \end{pmatrix} \simeq \begin{pmatrix} 0 \\ 0 \\ 2.2 \end{pmatrix}$$

Alice and Bob both use the tag "mountain" while Alice and Dave have the tag "sea" in common. But only Dave shares a resource with Alice, i. e. "photo 3". Consequently, the similarity between the feature vectors of Alice and Dave is higher than for Alice and Bob.

$$sim(v_{Alice}, v_{Bob}) = 0.29 \qquad sim(v_{Alice}, v_{Dave}) = 0.54$$

6.2 Managing Distributed Tagging Data

Centralized tagging systems like Flickr, YouTube, Delicious, and Bibsonomy are easy to use but they also have several drawbacks. A user has to sign up with different services to organize and share different media types, like Web links, photos, videos, and publications. Moreover, the data needs to be uploaded and stored within the infrastructure of the tagging service provider. This implies a certain loss of control over the data, a single point of failure, and the need to trust the service provider and to accept the respective terms of

service, e. g. Flickr limits the amount of photo storage space for free accounts (initially only 200 photos, now up to a certain volume) and also imposes a certain amount of censorship. Finally, with multiple tagging service providers it is hardly possible to connect the different resources types and obtain tagging statistics across system boundaries. Hence, it is currently not possible to use these systems for open data sharing, including other document types, like vector graphics, spread sheets, or ontologies.

Decentralized infrastructures are much more flexible and allow users to keep any kind of files locally while only exchanging the tagging meta data. An important advantage is the full control over the shared personal data without involving any central authority. Tagster [86] is a prototype of such a distributed collaborative tagging system. It has been implemented on top of a structured Peer-to-Peer network and offers the same basic tagging and search functionality as in centralized tagging applications (cf. screenshot of the Tagster user interface in Fig. 6.4). However, the computation of tag clouds and ranked retrieval is more complicated because of the data distribution and the cost for storing and maintaining all necessary statistical information.

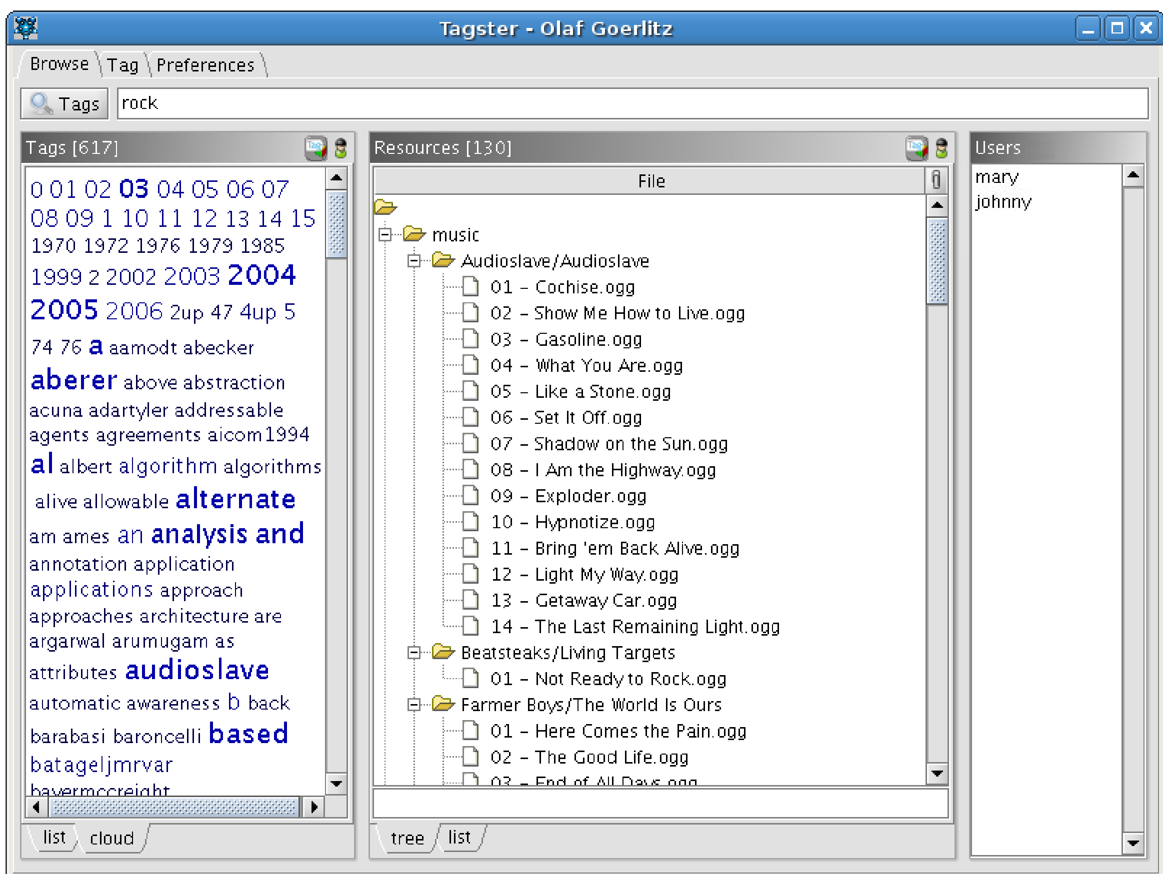


Fig. 6.4. Screenshot of the Tagster prototype. It combines file browser style resource navigation (hierarchical view) with collaborative tagging, i. e. tag-based browsing of local data and information shared by other users.

6.2.1 Decentralized Storage of Tagging Data

A peer (in Tagster) commonly represents one user. Therefore, it can answer all requests concerning the user's tagging data. But if global information is requested, e. g. the usage of a tag by other users, it is necessary to collect that data from the network. For example, the calculation of *if*-*iif*-based user-centric tag clouds would be highly inefficient if each peer needs to ask all other peers about the used tags. Instead, dedicated peers can maintain certain statistics such that (at best) only a few requests are necessary to retrieve the information. Therefore, the data distribution in a Peer-to-Peer system has a great influence on different non-functional aspects, like information retrieval performance, load balancing across peers, data locality, and flexibility and resilience in case of network changes. Different approaches for storing (RDF) data in a Peer-to-Peer network have been discussed in Sec. 3.6. Tagster is implemented on top of a *distributed hash table* (DHT), i. e. a structured overlay network, which has certain advantages over unstructured Peer-to-Peer systems. In a distributed hash table all data is represented as "key:value" pairs and indexing is done using a global consistent hash function which maps keys to peers. Thus, the hashing allows for an even distribution of the data across all peers and specific routing tables ensure that at most $\mathcal{O}(log\ n)$ routing steps are necessary to lookup any key in a network of n nodes (for more details see Sec. 3.6.2). A simple lookup-based interface provides access to all stored "key:value" mappings and each peer just maintains the data of its specific key range. No global knowledge about the data distribution is required. In addition, distributed hash tables are highly scalable, support flexible reassignment of key ranges, and provide good fault tolerance through data replication.

The storage of tag assignments in a distributed hash table requires a transformation of the tripartite tagging relations into "key:value" pairs. Therefore, the user-tag-resource combinations are split into the respective bipartite relations for tags and resources creating four different "key:value" mappings, i. e. "tag:user", "tag:resource", "resource:tag", and "resource:user". When put into the distributed hash table, each "key:value" pair is sent to the peer which is responsible for the respective key. For example, the assignment of multiple tags to one resource will create several "tag:resource" mappings which are indexed at different peers whereas all "resource:tag" mappings are stored at the same peer. In addition to the "key:value" mappings each peer also maintains statistical information for each stored key, i. e. *if* and *iif* values (cf. Sec. 6.1.3), which are used for the feature vector based relevancy ranking of results.

6.2.2 Feature Vector based Information Retrieval

Tag-based search and the browsing of associated users, tags, and resources are common interactions in folksonomies, which translate in Tagster into distributed hash table lookups that return all stored values for the specified key. Besides such basic operations, a user also wants to discover new and interesting

resources, e. g. through related topics or by finding users with similar interests. However, in order to compute the similarity between different items it is necessary to have a suitable data model that allows to characterize and rank items based on their respective tagging relations. Moreover, it must be implemented efficiently in a Peer-to-Peer network. Therefore, existing algorithms like Folkrank [117], which provides a resource ranking mechanism similar to Pagerank [181], cannot be applied because it focuses on centralized tagging systems where all data can be found and analyzed in a large database. In fact, the generic tag-cloud centered vector space model presented in Sec. 6.1.3 fits well for the Peer-to-Peer scenario since it does not need any complex calculations but solely relies on item counts.

In order to compute the *if* and *iif* values for the weights in a feature vector (as defined in Def. 6.4), a peer requires *local knowledge* about item-specific tag assignments and *global knowledge* about the occurrences of the item in tagging relations of all peers. For example, assuming that each user $u \in U$ acts as one peer in the overlay network and that, for ease of presentation, only tag-based user characterizations are considered, i. e. the coefficients in equation (6.14) are chosen accordingly to include only elements from the tag relations (e. g. $w_1 = 0$ and $w_2 = 1$). Then, the *if* value is the tag frequency for the user and the *iif* value is computed based on the overall number of users using this tag and the total number of users in the system. A peer can get the tag frequency from its locally stored data but it has to obtain the tag's *iif* value from the respective peer which is responsible for the tag in the distributed hash table. However, there are two challenges concerning the computation and exchange of *iif* values in a highly flexible Peer-to-Peer environment, i. e. 1) reliable counting of global item frequencies, e. g. total number of users, tags, and resources in the system and 2) updates of the *iif* values which are maintained by one peer but used in the feature vectors of other peers.

Counting in peer-to-peer systems is problematic because many peers may have relevant items but only a few should be contacted to gather the required information [176]. Moreover, for *if*-*iif* feature vectors it is necessary to have up-to-date total counts for each *iif* value. Although, it is possible to exploit the underlying network structure for cardinality estimations [115, 134] the tracking of a large number of cardinalities cannot be implemented efficiently. Alternatives are gossiping-based approaches [123, 130], where peers exchange counts iteratively until they converge to a stable value, sampling-based counting algorithms [24], or a combination of gossiping and sampling [155]. But these approaches do not scale for a large number of items in highly dynamic tagging scenarios or they cannot accurately estimate the number of infrequently used items. Only the probabilistic counting with distributed hash sketches [176] is suitable for Tagster in order to obtain the total number of users, tags, and resources from the distributed hash table. Consequently, all *iif* values are computed by the peers responsible for managing an item's tagging relations. The PINTS approach, which allows for efficient synchronization of the *iif* values with all other peers, will be presented in the next section.

6.3 Efficient Updates for Distributed Statistics

Global *iif* values are constantly changing as new tag assignments are being applied by different users of the distributed tagging system. Thus, local *if*-*iif* feature vectors can be influenced even if a user is not actively adding new tag assignments over a certain period of time. In order to avoid inconsistencies a user has to have up-to-date *iif* values whenever feature vectors are displayed (as tag clouds) or used for calculations. However, on demand update requests initiated by a user are typically very inefficient because many peers need to be contacted in order to retrieve all *iif* values for a feature vector. Moreover, some requests may even be unnecessary but a peer cannot know in advance if an *iif* value has been changed since it was last retrieved. Instead, PINTS follows an update propagation approach, i. e. users are notified about changes of the global *iif* values. But instant propagation of every *iif* update to all peers also leads to a high network communication overhead. Therefore, PINTS implements an update propagation strategy which decides dynamically when to notify peers. Hence, it is able to minimize the number of exchanged update messages while ensuring the accuracy of the local feature vectors at the same time. Following, the PINTS approach will be presented in detail.

6.3.1 Feature Vector Update Strategies

An update strategy for feature vectors has two orthogonal optimization goals. First, the propagation of information about changed global statistics, i. e. *iif* values, to all (potentially) affected peers in order to keep their local feature vectors accurate. Second, the limitation of the number of update messages, because the network communication overhead increases with the number of peers and for frequently used items, like popular tags. However, it is not possible to achieve both optimization goals at the same time. Thus, there is always a trade-off between the feature vector accuracy and the number of update messages. Following is a description of advantages and disadvantages of three different update strategies which either maximize the feature vector accuracy, minimize the number of update messages, or combine both optimization criteria to find the optimal solution.

Transient Updates. This naïve approach propagates every change of an *iif* value immediately to all affected peers. On the one hand this ensures that all local feature vectors are always up-to-date, but on the other hand it requires a lot of update messages, especially for items used by many peers. However, not every update will yield a significant change for a local feature vector. Especially for popular tags, there is a large number of associated peers while the weight of the propagated *iif* value is low and will have only little influence on the local feature vectors. Therefore, this update strategy is highly inefficient.

Interval-based Updates. The transient update approach sends significantly more messages than actually required for storing the respective data in the distributed hash table. In order to reduce the network communication cost updates can be limited by defining specific update intervals. Such an approach is useful if certain expensive computations, like item-based clustering or the analysis of popular trends, are only performed at specific times (e. g. daily at midnight). Meanwhile, the accuracy of the local feature vectors decreases because only the last observed *iif* value is maintained. However, *iif* values change at different rates. Hence, the accuracy of individual feature vectors can differ significantly. An individual tuning of update intervals can improve the accuracy but at the cost of additional management overhead.

Dynamic Updates. Changes of *iif* values can have different effects on the accuracy of feature vectors, e. g. depending on an item's popularity (rare items yield larger *iif* changes). Thus, the network communication cost can be minimized if update messages are only sent when necessary. However, dynamic update strategies come with an extra management overhead, i. e. for storing additional meta data and computing individual update conditions, in order to allow for a more flexible optimization and fine tuning of the trade-off between the number of update messages and the accuracy of the feature vectors. In fact, finding the optimal criteria for triggering update messages is challenging for different reasons. First, one needs to determine if a change of an *iif* value is significant for the accuracy of the affected feature vectors. Since this basically depends on the vector size and the other weights, it will be different for each feature vector. Therefore, update notifications are typically only necessary for a subset of peers. Moreover, the change rate of *iif* values is not stable over time. Hence, the use of individually but predefined update intervals is not feasible. Instead, the dynamic update strategy has to be based on continuous observations of the changes of *iif* values and feature vectors.

The approach implemented by PINTS follows the dynamic update strategy and additionally employs a prediction of changes to the global *iif* values based on discrete observations in the past. Thus, the number of update messages can be reduced even more because with accurate predications a peer can rely on locally available information and does not need to receive update messages as long as the predication is correct. But due to the dynamic nature of the tagging system, the predicted *iif* values will deviate from the 'true' values over time. Therefore, it is necessary to adjust the predictions as soon as the differences would have a significant effect on the feature vector accuracy. The problem, however, is to employ an efficient mechanism which allows index peers to detect deviations from the peers local predications and propagate updates on an individual basis. Therefore, PINTS implements a novel update approach using *evolution prediction* with *individual error approximation* which is described in the following section.

6.3.2 Prediction-based Feature Vector Approximation

For a locally maintained (user-centric) feature vector a peer in PINTS keeps track of the previously observed values $iif(i, \theta_1), \ldots, iif(i, \theta_n)$ at time points $\theta_1, \ldots, \theta_n$ and it predicts future values $iif^*(i, \theta)$ with a suitable approximation function. An investigation of the evolution of *iif* values for tags in Flickr and Delicious showed almost linear growth. Therefore, the approximation function is of the form

$$iif^*(i, \theta) = a_i \cdot \theta + b_i \tag{6.15}$$

where i represents an entry in the feature vector and the custom parameters a_i and b_i are derived through linear regression. Thus, the *iif* value of any feature weight can be predicted for arbitrary time points θ. Furthermore, the accuracy of an approximated tag cloud specific feature vector $v^*_{\mathcal{T}}(\theta)$ at time θ can be determined by computing the vector similarity between $v^*_{\mathcal{T}}(\theta)$ and the current "true" feature vector $v_{\mathcal{T}}$.

$$sim(v^*_{\mathcal{T}}(\theta),\, v_{\mathcal{T}}) > \delta \tag{6.16}$$

If the results lies above a predefined threshold δ (e. g. $\delta = 0.9$) there is no need to send an update message.

Propagating Updates

Every new tag assignment (u, t, r) is stored in the local database of peer u and sent to the index peers responsible for tag t and resource r, respectively. Each index peer updates its indexes, computes the new value for *iif*, and updates the approximation for $iif^*(\theta)$. The parameters of the approximation function are sent to the respective peers for use in their locally approximated feature vector $v^*(\theta)$. Assuming that the local predictions for all $iif^*(\theta)$ are correct, a compressed form of the estimated $v^*(\theta)$ is returned to the respective index peers in order to be able to determine if a new *iif* value might violate the accuracy of the peer's predicted feature vector.

However, large feature vectors, which are commonly encountered for the characterization of users with many tags, basically require to send the complete feature vector approximation to all respective index peers. Certainly, this is more information than an index peer needs for checking the "true" *iif* value with the approximation for one element of the vector. The rest of the approximated feature vector has no influence on the similarity computation in the context of one index peer. The following section explains details of the optimization used in PINTS which minimizes the amount of data that has to be exchanged between peers and index peers for communicating approximated feature vectors.

Compressed Approximation Parameters

An index peer, which receives new tagging data, will compute a new *iif* value and check if the accuracy of the approximated feature vector of any peer which relies on this *iif* value is changed beyond the predefined threshold (cf. the update condition in equation 6.16). Formally, a peer's approximated feature vector $v^*(\theta)$ and an index peer's variation of the approximated feature vector $v^*_m(\theta)$, where the m-th entry includes the index peer's *iif* value, are defined as

$$v^*(\theta) = \begin{pmatrix} if(i_1) \cdot (a_{i_1} \cdot \theta + b_{i_1}) \\ \vdots \\ if(i_m) \cdot (a_{i_m} \cdot \theta + b_{i_m}) \\ \vdots \\ if(i_n) \cdot (a_{i_n} \cdot \theta + b_{i_n}) \end{pmatrix} \qquad v^*_m(\theta) = \begin{pmatrix} if(i_1) \cdot (a_{i_1} \cdot \theta + b_{i_1}) \\ \vdots \\ if(i_m) \cdot iif(i_m) \\ \vdots \\ if(i_n) \cdot (a_{i_n} \cdot \theta + b_{i_n}) \end{pmatrix}$$

with values $a_{i_1} \dots a_{i_n}$ and $b_{i_1} \dots b_{i_n}$ representing the parameters for the linear approximation of the i-th element. However, the vectors differ only at the m-th element. Hence, the formula for the computation of the cosine similarity (equation 6.7) between the vectors contains identical terms for all other elements in both vectors.

$$sim(v^*(\theta), v^*_m(\theta)) = \frac{v^*(\theta) \cdot v^*_m(\theta)}{||v^*(\theta)|| \cdot ||v^*_m(\theta)||} = \frac{X}{Y \cdot Z} \quad \text{with}$$

$$X = \sum_{k \neq m} (if(i_k) \cdot (a_{i_k} \cdot \theta + b_{i_k})^2 \; + \; if(i_m)^2 \cdot (a_{i_m} \cdot \theta + b_{i_m}) \cdot iif(i_m),$$

$$Y = \sqrt{\sum_{k \neq m} (if(i_k) \cdot (a_{i_k} \cdot \theta + b_{i_k})^2 \; + \; if(i_m)^2 \cdot (a_{i_m} \cdot \theta + b_{i_m})^2},$$

$$Z = \sqrt{\sum_{k \neq m} (if(i_k) \cdot (a_{i_k} \cdot \theta + b_{i_k})^2 \; + \; if(i_m)^2 \cdot iif(i_m)^2}$$

The terms for the m-th vector item are isolated, i. e. $if(i_m)$, $iif(i_m)$, and $(a_{i_m} \cdot \theta + b_{i_m})$, such that the remaining sum expression, which is the same for X, Y, and Z, only depends on a peer's local *if* value and the linear approximation. By rearranging the sum formula around powers of θ, a quadratic polynomial is derived where the factors are represented by A_m, B_m, and C_m.

$$\sum_{k\neq m}(if(i_k)\cdot(a_{i_k}\cdot\theta+b_{i_k})^2 = \sum_{k\neq m} if(i_k)^2\cdot(a_{i_k}^2\cdot\theta^2+2a_{i_k}b_{i_k}\cdot\theta+b_{i_k}^2)$$
$$= \sum_{k\neq m}[if(i_k)^2 a_{i_k}^2\cdot\theta^2+2if(i_k)^2 a_{i_k}b_{i_k}\cdot\theta+if(i_k)^2 b_{i_k}^2]$$
$$= A_m\cdot\theta^2 \;+\; 2\,B_m\cdot\theta \;+\; C_m \quad \text{with}$$

$$A_m=\sum_{k\neq m} if(i_k)^2 a_{i_k}^2,\quad B_m=\sum_{k\neq m} if(i_k)^2 a_{i_k}b_{i_k},\quad \text{and}\quad C_m=\sum_{k\neq m} if(i_k)^2 b_{i_k}^2.$$

This aggregation of terms allows for a very compact representation of all values necessary for the similarity computation between $v^*(\theta)$ and $v^*_m(\theta)$. Only seven distinct parameters $[A_m,\, B_m,\, C_m,\, if(i_m), iif(i_m), a_{i_m}, b_{i_m}]$ are needed to capture all local approximations and the globally known values. A peer basically computes A_m, B_m, and C_m for its feature vector weights and sends the values to the index peers responsible for i_m. The index peer knows $if(i_m)$ and $iif(i_m)$. Moreover, it keeps track of the previous evolution approximation parameters a_{i_m} and b_{i_m} which were propagated to the peers. Thus, it holds all required parameters to compute the similarity between the two feature vectors, and it can check the consistency condition (Eq. 6.16) whenever the global values change. Update messages to peers are only exchanged if the vector similarity falls below the specified threshold. In that case, an index peer will notify all affected peers about the new approximation and it receives in return an updated set of the parameters A_m, B_m, and C_m.

The space requirements for storing all parameters at an index peer is small and it depends on the actual number of peers which use the specific item i_m. In general, the index peer has to store the aforementioned seven values for each peer. With 8 *bytes* per value (i. e. long integers or double precision floating point numbers) an entry is 56 *bytes*. In addition, maintaining a history of up to five previous *iif* values including the corresponding timestamps requires 80 *bytes*. For example, the global tag-specific index for a large snapshot of the Flickr dataset [8], with about 320,000 users, 1.6 million tags, and an average of 50 user per tag, would require a total space of 4.6 GB. Since the indexes are distributed across many index peers, each of them only has to store a little amount of data, e. g. 4.6 MB for 1000 index peers. Hence, in a Peer-to-Peer network, where each user also acts as an index peer, the storage size for keeping the statistics is not significant compared to the actual tagging data maintained by each peer.

[8] http://west.uni-koblenz.de/Research/DataSets/PINTSExperimentsDataSets

6.4 Evaluation

The objective of the evaluation is to show that PINTS is actually able to significantly reduce the network communication overhead while maintaining a high accuracy of the local feature vectors. Therefore, a systematic comparison was done between the three discussed update strategies (cf. Sec. 6.3.2), i. e. transient updates, interval-based updates, and the PINTS approach. The relevant measures for the evaluation are the number of update messages transmitted over the network and the number of similarity threshold violations for local feature vectors before updates with the correct *iif* values were propagated. The first measure is an important performance indicator in Peer-to-Peer systems while the latter captures the effectiveness of the compared update strategies.

In order to obtain authentic results, the evaluation exploits real tagging data from two of the earliest and most popular social tagging platforms, i. e. Delicious and Flickr. By replaying each tag assignment from these datasets the evaluation can simulate real tagging activity of different users. The main difference between Delicious and Flickr is the correlation between users and resources. The photos in Flickr are only tagged by one user whereas Delicious users can add tags to any resource. Both tagging datasets were obtained from the Tagora research project[9] which crawled the Delicious and Flickr web sites over a longer period of time. Each dataset contains over a hundred million individual tag assignments from several hundred thousand users using more than a million different tags (cf. Table 6.1). A tag assignment is a four-

Table 6.1. Statistics of the Flickr and Delicious tagging datasets

	users	tags	resources	tag assignm.	time span
Flickr	319,686	1,607,879	28,153,045	112,900,000	Jan 04 - Jan 06
Delicious	532,924	2,481,698	17,262,480	140,126,586	Jan 03 - Jan 06

tuple which consists of time stamp, user ID, tag ID, and resource ID. The datasets were anonymized due to legal issues concerning the republication of the crawled data. But this is irrelevant for the evaluation which only requires that the original tag assignments are available. Next, the evaluation setup is described. Then, detailed results will be presented.

6.4.1 Setup

Replaying all tag assignments of the Flickr and Delicious datasets in a real Peer-to-Peer system is complicated because global data consistency and the correct event order has to be ensured. Therefore, this evaluation employs

[9] http://www.tagora-project.eu/

PeerSim [165] to simulate a large Peer-to-Peer network in a controllable environment. Each individual user from the datasets is modeled as a peer which stores all tag assignments of the user locally and also sends notifications to the respective index peers, which maintain the global statistics. The number of index peers in the distributed hash table overlay network is set to 1000. PeerSim provides an event-based simulation framework, i. e. all tag assignments are processed in time stamp based order as discrete events. This allows for replaying a complete stream of tag assignments and performing the appropriate global statistics computations and sending the necessary update messages. Although the time stamp information in the tag assignments is irrelevant for the baseline update strategy and for the PINTS approach it is needed for triggering the update events in the interval-based update strategy and for the time-based comparison of the evaluation results.

6.4.2 Results

The evaluation results are presented individually for Flickr and Delicious in order to show differences in their characteristics. In general it can be observed that the popularity of both platforms increased significantly during the evaluated time frame. This is visible through a larger number of tag assignments which also effects the exchanged messages and the rate at which violations of the vector similarity occur.

Similarity Threshold Violation

The first comparison assesses the ability of the different update strategies to maintain the accuracy of the local feature vectors, i. e. by computing the cosine similarity with the true feature vector (Eq. 6.16). Interval updates were performed at a rate of one day and five days. The number of local feature vectors with a violated accuracy was determined just before the next interval update was applied. A feature vector similarity above 90% was considered acceptable. Hence, the threshold was set to $\delta = 0.9$. For the PINTS approach the same threshold value was used as the tuning parameter and, in the same way, the number of diverging vectors was computed before a local vector was updated (which may occur at any time). Figure 6.5 shows the average number of violated feature vectors for Flickr and Delicious, normalized by the total number of users (peers) which had been encountered so far. The transient update strategy is not shown in the diagrams since it does not yield any similarity violations.

There is some fluctuation in the curves, especially in the beginning when the tagging activity in certain intervals can significantly influence the global *iif* values, e. g. through bursts of new tag assignments or new topics and trends. As expected, the curve of the five day update interval lies well above the curve of the one day update interval, since the deviation is higher with five times less updates. The results for PINTS improve significantly after a

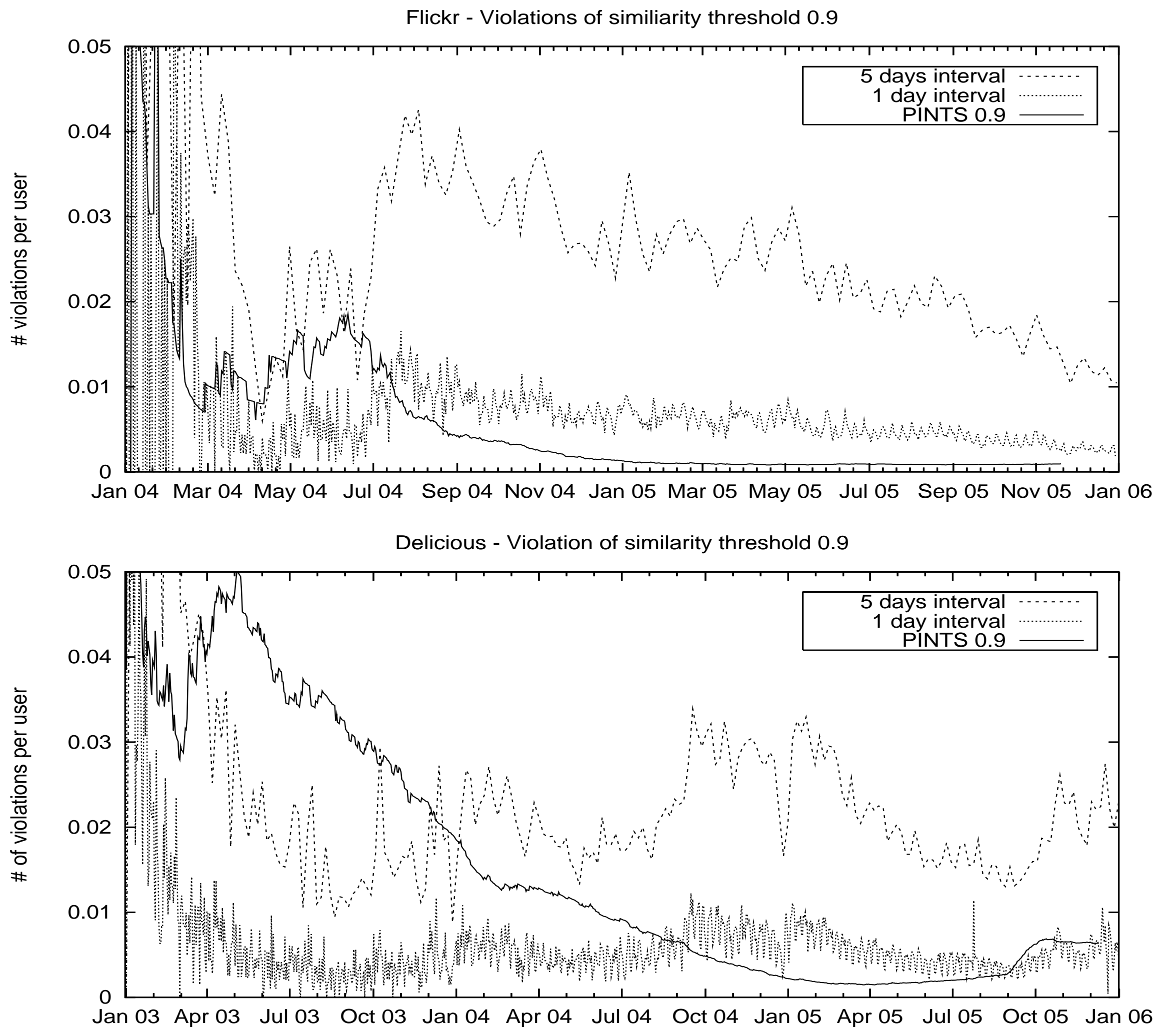


Fig. 6.5. Relative frequency of violations of the similarity threshold $\delta = 0.9$ with different update strategies for Flickr (above) and Delicious (below)

first period of instability, i. e. changes by individual tag assignments have less effects due to a larger number of previous tag assignments which dominate in the computation of the weights. Thus, changes in the global *iif* values are smaller and the approximations are more accurate. Concerning the difference between Flickr and Delicious, one can see that the PINTS curve stabilizes much quicker for Flickr and remains at a very low value. This is certainly an effect of the specific tagging restrictions of Flickr which allow only one user to tag a resource. In Delicious there is more freedom in tagging resources which makes tag assignments less predictable. Two more notable observations are the burst in Flickr around July 2004 and in Delicious in September 2005. While in all cases the measures for the interval updates get worse, it only affects the PINTS approach in Flickr. For Delicious there is no noticeable change.

Message Complexity

Figure 6.6 shows the number of messages for transmitting changed *iif* values by the different update strategies. Due to the rapidly increasing number of tag assignments, all values have been normalized by the total number of tags at that specific point in time. Thus, the graphs show the average number of update messages per tag which are sent by an index peer.

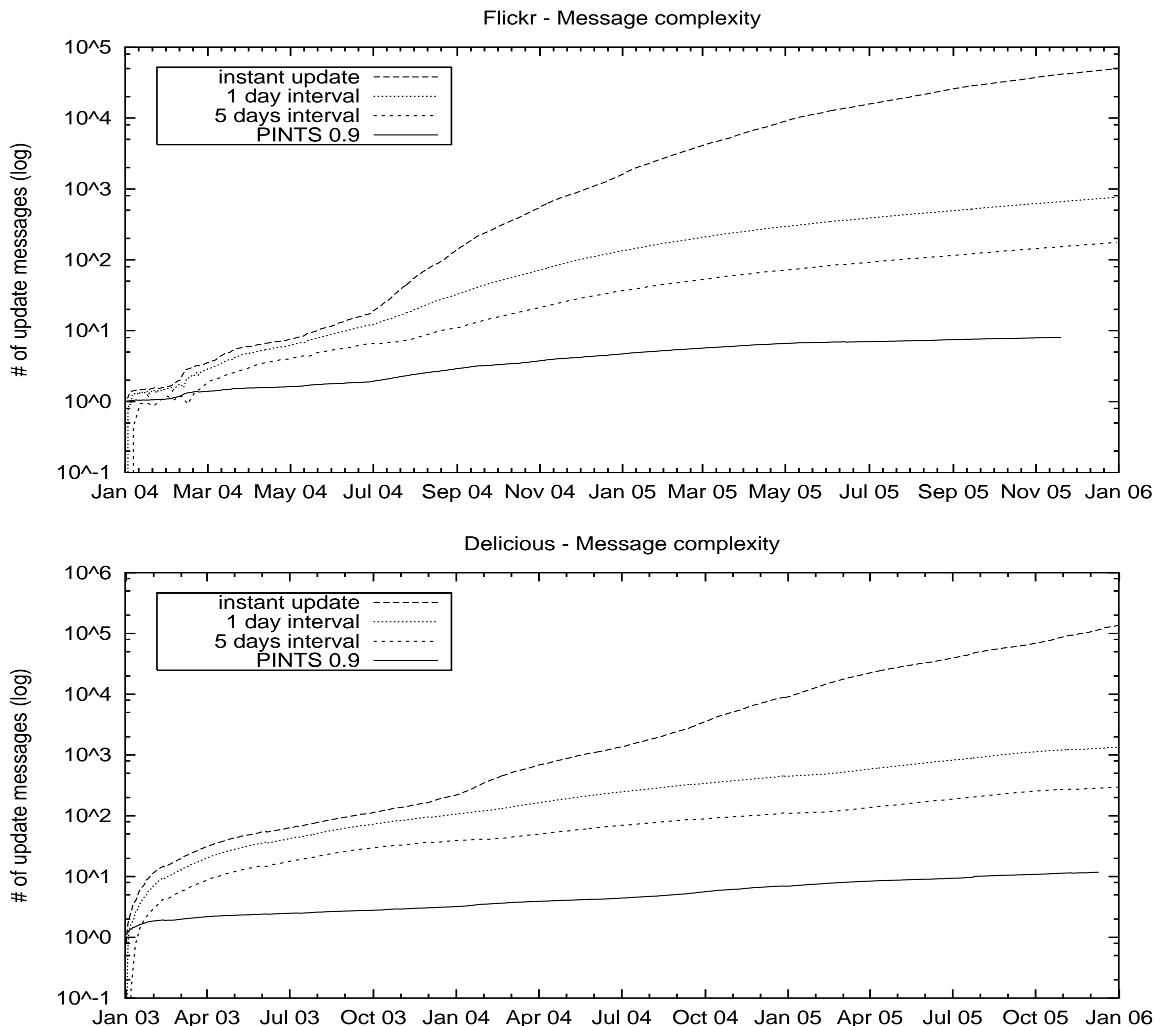


Fig. 6.6. Message complexity for different update strategies for Flickr (above) and Delicious (below).

The transient update strategy sends the largest number of update messages. In addition to the fast growth (note the log scale) it also highlights the points in time where Flickr and Delicious experienced a boost of new tag assignments. Compared to the other approaches the values for the transient update are by orders of magnitude higher. For the interval based updates one can see how much a shorter update interval increases the number of messages.

In contrast, the PINTS approach clearly employs the lowest number of update messages, i. e. only one to ten messages per index peer and tag.

Discussion

The evaluation shows that the PINTS approach fully meets the expectations. A reduction of the number of update messages can be achieved with only little sacrifices concerning the accuracy of the local feature vectors. Compared with the interval-based approaches, PINTS can ensure the same level of accuracy like daily updates but with two orders of magnitude less messages. However, in the initial phase the accuracy of the locally maintained vectors is generally problematic for all update strategies since the feature vector weights can change quickly through a few new tag assignments. In order to overcome this problem PINTS can be flexibly tuned with a higher similarity threshold. As soon as the approximations become more stable a lower threshold can be applied. The interval-based update strategies do not allow for such a kind of flexibility. Hence, a manual configuration would be required.

6.5 Summary

Information retrieval on distributed, graph-structured data is an interesting research area with different application scenarios. This chapter focused on a decentralized collaborative tagging architecture where large amounts of highly connected data are stored in a structured Peer-to-Peer network. However, such a Peer-to-Peer infrastructure usually supports only lookup operations. In order to allow for result ranking and similarity-based searches a generic tag-cloud model was defined. It captures different aspects of the tripartite tagging relations and is basically an adaption of *tf-idf* feature vectors which are commonly used in traditional information retrieval approaches. However, the maintenance of statistical data in a distributed system is challenging, especially when frequent data updates occur. For example, the combination of global and local statistics in the aforementioned tag-cloud data model requires to exchange information between index peers and client peers whenever the data is changed. Therefore, the presented PINTS approach implements a novel and efficient update propagation strategy for global statistics. It employs an evolution-based approximation model for global meta data and ensures up-to-date local feature vectors with only few update messages. The overhead for storing the additionally information for the approximation coefficients is minimal compared to the actual data. Furthermore, an evaluation has shown that PINTS can drastically reduce the network communication overhead while still achieving a high accuracy for the approximated local feature vectors. Moreover, PINTS is flexible with respect to the application scenario and can be adapted for other domains with similar data structures.

7

SPLODGE Benchmark Methodology for Federated Linked Data Systems

Benchmarking is traditionally used to compare different software implementations according to specific performance metrics. It has a long tradition in the database community [45, 49, 66] where state-of-the-art research systems and commercial databases are compared against each other. Thus, benchmarks usually cover a wide range of application scenarios, i. e. common requests as well as challenging corner cases, in order to obtain a comprehensive picture over the capabilities of the tested systems. The benchmark handbook by Gray [90] mentions four general requirements for a benchmark, i. e. it has to be 1) *relevant*, i. e. test and measure the typical operations in the problem domain, 2) *portable*, e. g. run on different systems and architectures, 3) *scalable*, thus applicable on small and large computer systems, and 4) *understandable* in order to be credible.

RDF benchmarks [92, 199, 32] have been designed with essentially the same objectives, i. e. to compare RDF triple store implementations by measuring their performance for different queries of varying complexity. However, query processing on RDF data is different in certain ways from the query processing on highly structured relational databases. For example, RDF data is less structured, has a high degree of data correlation, and many datasets do not provide explicit schema information [192]. Moreover, new challenges are introduced by federated query processing on Linked Data which has received a lot of attention recently. Compared to (distributed) relational database systems, which have matured over the past decades, the research on efficient query evaluation across RDF data sources is still in its infancy [190]. Hence, benchmarks for federated SPARQL query processing are necessary in order to evaluate such systems, taking into account the peculiar aspects as data distribution and connections between data sources in the Linked Data cloud.

But the choice of available RDF federation benchmarks is limited. Currently, only FedBench [198] offers a complete benchmark suite, including a variety of federated SPARQL queries for a selected number of linked RDF datasets. Hence, it has already been used by other researchers and it is also employed to benchmark SPLENDID in Chap. 4 and 5. However, FedBench (as well as other federated benchmark proposals) typically have certain limitations, e. g. with respect to scalability, flexibility, and complexity of the benchmark queries. Therefore, capturing dataset and query characteristics which are typical for Linked Data is still a challenge.

This chapter starts with a discussion of design issues for federated RDF benchmarks. Therefore, several currently available RDF benchmark approaches are surveyed with respect to their strengths and weaknesses. Then, a classification of common benchmark query characteristics is presented which also takes into account aspects for benchmarking federated systems. Afterwards, details of the SPLODGE [89] benchmark methodology are presented. Based on findings from the query characterization and it allows for flexible generation of random benchmark queries. Finally, an evaluation of the SPLODGE benchmarking methodology on real Linked Data concludes the chapter.

7.1 Design Issues for Federated RDF Benchmarks

In contrast to centralized RDF benchmarks a benchmark for federated SPARQL query processing systems should 1) be based on characteristic linked RDF datasets, with various schemas, different levels of structuredness, and small to large data sizes, 2) provide typical Linked Data queries, having different graph patterns and query constructs with varying complexity, and 3) consider an evaluation environment which incorporates physical characteristics and constraints of the Linked Data cloud, i. e. network latency, limited bandwidth, and restrictions of SPARQL endpoints. However, existing benchmarks hardly satisfy all these requirements. From the currently available set of RDF benchmarks the majority [92, 199, 32, 168, 167] is designed for centralized triple stores. So far, FedBench [198] is the only benchmark which is exclusively designed for distributed query processing scenarios. However, it has limitations as well, e. g. it does not allow for large-scale evaluations.

7.1.1 Datasets

A typical problem for designing RDF benchmarks is the choice between artificial and real datasets. Especially in the context of Linked Data they both have certain advantages and disadvantages. In general, the former are ideal for modeling complex benchmark queries while the latter allow for more realistic evaluation scenarios. Artificial datasets are typically found in centralized RDF benchmarks, e. g. in LUBM [92], BSBM [32], and SP^{2}Bench [199]. Examples for benchmarks utilizing real RDF datasets are DBPSB [168, 167], FedBench [198], and LIDAQ [219].

Artificial Datasets

The *LeHigh University Benchmark* [92] (LUBM) was among the first benchmarks for evaluating RDF triple stores. Its data model is centered around university resources, e. g. defining relations between students, lecturers, and courses. A dataset generator can create synthetic benchmark datasets of arbitrary sizes. LUBM provides hand-crafted queries which cover complex query expressions, different result sizes and also include inferencing of subclass relations. Thus, the performance evaluation includes several challenging queries. The *Berlin SPARQL Benchmark* [32] (BSBM) focuses on an e-commerce use case, i. e. the benchmark queries are centered around the search and navigation of consumers who are looking for specific products. The data schema describes products with different features, offers, and consumer reviews. Again, a data generator allows to create artificial datasets of arbitrary size. The benchmark queries include different SPARQL query operators and result modifiers. Different query structures are used to cover diverse query processing aspects. The *SPARQL Performance Benchmark* [199] (SP^{2}Bench) generates artificial datasets which follow the DBLP publications schema [141]. Its data generator creates arbitrarily large datasets with the characteristic data distribution and data correlations of DBLP. The hand picked SPARQL benchmark queries represent meaningful requests on the data. In order to test a broad range of different query types, a variety of operator combinations, including `UNION`, `FILTER`, and `OPTIONAL`, with different RDF access patterns are covered.

These artificial RDF datasets provide a well controlled environment to design complex SPARQL queries for testing various aspect of RDF triple stores. Individually crafted data schemas help to realize the desired RDF graph characteristics which are required by specific benchmark queries. Since artificial datasets are typically highly structured [70] they can hardly resemble the characteristics of real datasets from the Linked Data cloud. The latter are very diverse with respect to schema and structuredness. Moreover, artificial datasets can hardly be partitioned in a suitable way to represent a Linked Data federation scenario with appropriate links between resources. Finally, the benchmark queries are missing typical aspects related to distributed data retrieval in the Linked Data cloud. Hence, a federated RDF benchmark should ideally be based on real data from the Linked Data cloud.

Real Datasets

The Linked Data cloud contains a large variety of RDF datasets with different characteristics (cf. Sec. 2.2.2). Hence, a good set of benchmark datasets should typically cover different dataset sizes, structuredness, schema complexity, data quality, and links to other data sources. Moreover, the access methods, i. e. URI resolution, SPARQL endpoint, or download of data snapshots, and the frequency of updates may also be important because benchmark results should be reproducible and comparable.

The *Benchmark Suite for Federated Semantic Data Query Processing* (FedBench) [198] is currently the only widely accepted federated RDF benchmark framework which integrates diverse data sources from the Linked Data cloud, including DBpedia [15, 31], which is by far the most prominent dataset, and others, like Drugbank, ChEBI, or Musicbrainz. The benchmark queries have been hand-picked and cover different aspects, like star and path join patterns as well as different solution modifiers. Moreover, they are divided into distinct query sets which target different data domains (e. g. life science, cross domain, and a partitioned SP^{2}Bench dataset). However, there are also limitations of FedBench, e. g. the fixed number of ten preselected datasets and that each query involves no more than five of them. LIDAQ [219] implements a more scalable benchmarking approach which is based on crawled Linked Data. It automatically generates query sets by employing random walks on the RDF graph. Two different query pattern structures are supported, i. e. resource-centric (star-join) and path queries. Variations are applied by the substitution of RDF terms with variables in the query patterns. Other query operators besides conjunctive joins are not considered. The random walk approach ensures that there exist at least one result for any join pattern combination. But it is not possible to control the result size or the number of data sources which are involved in the query evaluation. The *DBpedia SPARQL Benchmark* (DBPSB) [168, 167] analyzes real queries from the official query log of the DBpedia SPARQL endpoint. Normalization and clustering is applied to identify common query characteristics. The results are 25 queries which cover different SPARQL features, e. g. `JOIN`, `UNION`, and `OPTIONAL`, solution modifiers, and filter conditions. Different parts of these queries can be varied randomly. Moreover, a data generator allows to compute subsets of DBpedia and maintain the characteristic data distribution.

Using real linked datasets allows for designing authentic federated benchmarks. But the examples above also illustrate some problems. A manual selection of data sources can be tedious, especially if different linked dataset characteristics should be covered. Moreover, the benchmark scalability is limited because of fixed dataset sizes and difficulties with dataset partitioning. In contrast, a crawling-based approach is much more flexible since it can produce Linked Data corpora of any size with the common characteristics. However, it is often difficult to understand which data is actually collected. Hence, the generation of meaningful benchmark queries becomes much more complicated.

7.1.2 Queries

The choice of queries is important for a benchmark because the benchmark's usefulness and credibility depends on the coverage of typical application scenarios. Thus, an important objective is to find suitable benchmark queries for a given set of Linked Data sources. But since federation systems are not in wide use yet, there exist no query logs which could be used to identify common distributed queries across multiple RDF data sources. The bench-

marks presented in the previous section follow different approaches to obtain suitable benchmark queries (cf. Tab. 7.1 for detailed features). FedBench employs hand-picked SPARQL queries which resemble meaningful requests on the selected data. But this manual approach requires a good knowledge about dataset schemas and links. Moreover, the size and diversity of the Linked Data cloud makes it almost impossible to cover all potentially relevant aspects with hand-picked queries. LIDAQ in turn, creates random queries according to a predefined set of basic query structures. Thus, it allows for more flexibility and better coverage of the datasets. But the generated SPARQL queries cover only a limited number of typical query characteristics and it remains unclear if they are realistic in a federation scenario. Therefore, the goal of SPLODGE is to overcome this gap with a benchmark methodology that offers automatic generation of complex SPARQL queries for arbitrarily large RDF datasets.

Table 7.1. Overview of typical query features covered in RDF benchmarks. For each feature the number of queries which contain the feature are given.

	LUBM	SP^{2}Bench	BSBM	FedBench	DBPSB	LIDAQ
use case	university	publication	e-commerce	Linked Data	DBpedia	Linked Data
dataset	synthetic	synthetic	synthetic	real (+SP^{2}B)	real	real
data size	variable	variable	variable	fixed	variable	variable
#queries	14	12	12	$7+7+11$	25	random
max joins	5	12	14	6	11	3
var pred.	-	1	1	2	5	-
union	-	2	2	3	10	-
optional	-	3	4	1	8	-
filter	-	6	9	1	15	-
negation	-	2	1	-	2	-
limit/offset	-	1/1	6/1	-	(25)/-	-
order by	-	2	6	-	-	-
distinct	-	5	3	-	13	-

7.1.3 Evaluation Environment

The main purpose of benchmarks is to measure specific performance metrics which can be used to compare different implementations. However, for a federation benchmark this does not only depend on the choice of datasets and queries but also on the actual benchmark setup. For example, live queries on SPARQL endpoints are not feasible for reliable benchmark results because of specific endpoint restrictions, network latency, and varying bandwidth. Therefore, snapshots of Linked Data sources are often used in a controlled benchmark environment. Moreover, it must be possible to acquire different performance metrics in an easy way. Besides query execution time, which can

be differentiated between response time for returning the first results and the total execution time, there are other aspects which are typically interesting for a federation benchmark.

Transmitted Data The distributed SPARQL query evaluation requires to send messages to designated SPARQL endpoints. Depending on the query execution strategy, the amount of data transferred over the network can differ significantly. The effects of an optimization focusing on minimization of the communication cost can be measured with respect of the size of the data which has been transmitted.

Result Completeness Obtaining complete results may require many requests to different data sources. However, in order to prevent long query execution times a federation implementation may contact only a subset of the data sources or set a timeout for the query execution. Hence, query execution time should not be the only performance measure but result completeness has to be taken into account as well.

CPU/Memory Utilization Federated databases [206] support cooperation between different data sources to a certain degree, i. e. the mediator delegates query execution tasks among the participating nodes, e. g. by shipping data. Limitations of SPARQL, however, leave much of the query processing responsibility to the mediator. Thus, RDF federation benchmarks may also consider CPU and memory utilization as performance criteria.

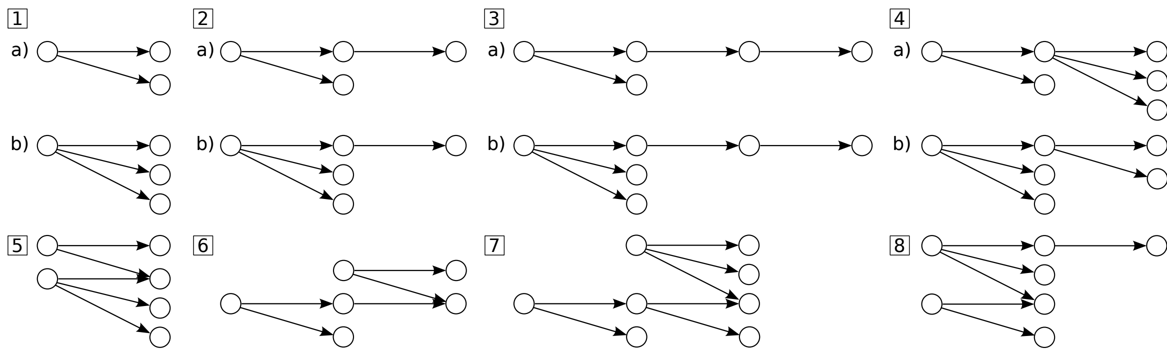
7.2 SPARQL Query Characterization

The objective of SPLODGE is to generate random SPARQL queries for real Linked Data sources including certain characteristics that are specifically suited for evaluating federated SPARQL processing approaches. Thus, in order to provide a suitable set of benchmark queries, i. e. from common cases to border cases, it is necessary to identify appropriate query characteristics for a federation benchmark. Centralized benchmarks with artificial datasets [92, 199, 32] already cover a wide range of typical SPARQL query features. But these need to be extended such that the distribution of related entities across different datasets is taken into account, i. e. queries across multiple Linked Data sources are of major interest. However, identifying common features for federated queries is complicated because there exist no publicly available federation systems which could provide query logs. Hence, there is a general need for a characterization of typical (federated) benchmark queries in order to allow for defining the desired query features.

Following summarization of query characteristics is based on a collection of findings from query log analysis of centralized RDF stores [162, 184, 82] and the investigation of typical queries used in the aforementioned benchmarks, e. g. FedBench [198] which is currently one of a few benchmarks which provides meaningful hand-picked queries spanning multiple Linked Data sources.

The query characterization is divided along three dimensions, namely *Query Algebra*, *Query Structure*, and *Query Cardinality*. These dimensions and features of the query characterization are considered representative to define a large set of different benchmark queries. However, it is not assumed to be complete. With an advanced development and use of federated systems, it is possible that more characteristics may be identified.

The FedBench [198] queries were used as a reference for the analysis of common query structures. Figure 7.1 gives an overview of the triple pattern structure of the life-science, cross-domain, and linked data queries. The visualization is focused on conjunctive joins. Filters and optional parts are omitted. Variable names and bound RDF terms are left out as well in order to highlight the graph structure. Thus, different combinations of triple patterns in star-joins and path-joins can be clearly seen. Queries 1 - 3 contain subject-based star-joins, in the case of the latter two combined with a path-join. Queries 4 and 5 combine two star joins, one with a subject-object join, the other as an object-object join. The last two query structure types 7 and 8 combine multiple star-join patterns in different ways with a path-join.



	1	2	3	4	5	6	7	8
	star	path + star		star + star		path + star*		
Life Science			ls3		ls6-7		ls4	ls5
Cross Domain			cd5-7	cd3		cd4		
Linked Data	ld5,7	ld1-2,9-11	ld3					ld4

Fig. 7.1. Overview of different graph pattern combinations (path/star) found in the FedBench queries (top). The life-science (ls1-7), cross-domain (cd1-7), and linked data (ld1-11) queries are differently distributed among the query graph patterns (below).

7.2.1 Algebra

The first set of query characteristics relate to the semantic properties of SPARQL. Different query operators are supported by SPARQL. Research on the query complexity [183, 200] has shown that SPARQL query evaluation beyond Basic Graph Pattern, i. e. conjunctive joins and filter expression, is PSPACE-complete and that the main source of complexity is the `OPTIONAL` operator.

Query Form. SPARQL has four query forms, i. e. `SELECT`, `CONSTRUCT`, `ASK`, and `DESCRIBE`. Query evaluation is based on matching graph patterns as defined in the `WHERE` clause of a query. Solutions are returned as a multiset of variable bindings (`SELECT`), an RDF graph constructed from triple templates with substituted solution bindings (`CONSTRUCT`), a boolean value indicating the existence of solutions for the graph pattern (`ASK`), or a graph describing the resources that have been matched (`DESCRIBE`). Instead of a graph pattern `DESCRIBE` queries can also take a single URI.

Join Type. Joins in SPARQL are defined by combinations of triple patterns. Three different join types are supported, i. e. *conjunctive join* (`.`), *disjunctive join* (`UNION`), and *left-join* (`OPTIONAL`). It has been shown [200] that the `OPTIONAL` operator significantly increases the complexity of the query evaluation.

Result Modifiers. The results of the pattern matching, so called solution sequences, can be altered by modifiers like `DISTINCT`, `LIMIT`, `OFFSET`, and `ORDER BY`. The application of these additional constraints also increases the complexity of the query evaluation.

7.2.2 Structure

The next properties deal with the graph structure that is defined by the triple patterns in a query's `WHERE` clause. There are basically two aspects for variability in the query structure, i. e. the assignment of variables to subject, predicate, or object position in a triple pattern and the connection of triple patterns via join variables.

Variable Patterns. Eight different combinations are possible for having zero to three variables in subject, predicate, or object position of an RDF triple pattern. Not all combinations are equally common, e. g. triple patterns with unbound predicate are rarely used while a bound predicate and variables in subject and/or objection position occurs quite often.

Join Patterns. Triple patterns with a common variable define an equi-join over the solutions of the triple patterns. Queries can have different graph shapes depending on the position of the join variable in the triple patterns. Typical join combinations are subject-subject joins (star shape) and subject-object joins (path shape). The combination of star-shaped and path-shaped joins yields hybrid join patterns. (cf. Fig. 7.1)

Cartesian Product. The graph structure of joined triple patterns may have disconnected parts if they do not share a common join variable. In such a case the result is the cross product of the partial solutions. The disconnected graph patterns can be evaluated independently. However, the result set of a Cartesian product may be quite large which implies a large overhead for the query evaluation.

7.2.3 Cardinality

The third group of properties deals with the number of sources and joins involved in the evaluation of a query and with the size of the returned result sets. Thus, the group includes specific aspects relevant for distributed query execution.

Number of Relevant Sources. Federated queries are intended to span multiple Linked Data sources. Therefore, the number of *relevant data sources*, i. e. the data sources which match the graph patterns of a query, is an important property. Let $\mathcal{P}$ be a graph pattern and let $\mathcal{F}$ be a federated RDF graph. Then the set of relevant data sources is

$$\mathcal{F}_{\mathcal{P}} := \{\mathcal{G} \in \mathcal{F} \mid \exists\, \mathcal{P}' \in \mathcal{P} : eval(\mathcal{P}', \mathcal{G}) \neq \emptyset\,\}.$$

Number of Joins. The complexity of a query increases with the number of joined triple patterns because the query optimizer has to consider a larger space of possible query execution plans. Moreover, the cost for query processing increases as well, e. g. due to a larger number of sub queries and the increased amount of data transferred over the network. The number of joins of a graph pattern $\mathcal{P}$ are defined based on the number of contained triple pattern $\mathcal{T}$.

$$joins(\mathcal{P}) = |\{\mathcal{T} \in \mathcal{P}\}| - 1$$

Result Size. The number of tuples in the solution sequence define the result size of a query. However, different data sources can contribute a different number of result tuples. Thus, the final result set is actually a combination of the source-specific results and the *query selectivity* is used to define the proportion between the overall number of triples in the relevant data sources and the number of triples which actually contribute to the query result. The evaluation of a graph pattern $\mathcal{P}$ on a federated RDF graph $\mathcal{F}$ yields following selectivity.

$$sel_{\mathcal{F}}(\mathcal{P}) = \frac{\sum_{\mathcal{G} \in \mathcal{F}_{\mathcal{P}}} |eval(\mathcal{P}' \in \mathcal{P}, \mathcal{G})|}{\sum_{\mathcal{G} \in \mathcal{F}_{\mathcal{P}}} |\mathcal{G}|}$$

7.3 Query Generation Methodology

The SPLODGE query generation approach, which focuses on creating random SPARQL queries for Linked Data sources, provides a methodology and a toolkit that can be used to define federation benchmarks for evaluating RDF federation systems in a reproducible and comparable manner. Therefore, three steps are necessary, namely (1) *query parameterization*, (2) *query generation*, and (3) *query validation*. The query parameters define the characteristics of the query which should be generated. The query generation is an automatic process which creates random queries according to the query parameters. Finally, the query validation ensures that all query constraints are met. A more detailed explanation of these three parts is given below.

7.3.1 Query Parameterization

The SPLODGE query generator is based on parameterized query descriptions which define structure, complexity, and cardinality constraints according to the characteristics described in Sec. 7.2. Such query parameterizations allow for defining different types of benchmark queries in a very flexible way and to create random queries of the same type.

The query parameterization is primarily oriented on the query structure, i. e. the definition of basic graph patterns with the desired constraints. Through the combination of path-join patterns and star-join patterns all possible variations of basic graph patterns can be constructed. Path-joins and star-joins are defined by the number of joined triple patterns. This also specifies the cardinality constraint for the number of triple patterns in a query. Another parameter defines the number of data sources which should be involved in producing results for the join pattern. However, this parameter has different semantics for path-joins and star-joins. For path-joins, it defines the minimum number of linked data source to be involved such that the path-join can be evaluated across them. Hence, it cannot be greater than the number of joined triple patterns. For star-joins it refers to the number of data sources which can individually match all triple patterns of the star-join. The combination of different join-patterns is defined via *anchor nodes* which have to be in subject or object position, e. g. nodes (?*a*), (?*b*), and (?*c*) in Fig. 7.2 are anchor nodes. Joins across predicates are currently not supported.

By default, all predicates in the triple patterns are assumed to be bound while subjects and objects are assumed to be unbound variables. The result are common schema-level queries. However, the parameterization also allows to define that a number of predicates should not be bound or that a subset of subjects and objects will be bound. In some cases it may be desirable that the value for bound variables can be controlled. Therefore, the query parameterization may define specific predicates, e. g. owl:sameAs, or specific class types, like foaf:Person, to be included as bound RDF terms of a query. These value restrictions are limited to schema-based URIs.

7.3.2 Iterative Query Generation

Based on a given query parameterization the query generator creates random queries, i. e. queries with the desired graph structure and algebraic complexity where RDF terms for bound variables in the triple patterns are chosen randomly from the set of eligible values. Thus, the result is a set of queries with the same structure but different result sets. Figure 7.2 illustrates a query with a path-join and attached three star-join patterns. The triple patterns contain the (join) variables $?a$ to $?h$, and randomly assigned predicates p_1 to p_7. Path-joins and star-joins are defined by using the same variable in subject or object position of different triple patterns. The SPLODGE query generator creates triple patterns which reuse variable names and have constant RDF terms only in predicate position. This restriction simplifies the query generation and the resulting queries are based on schema information only.

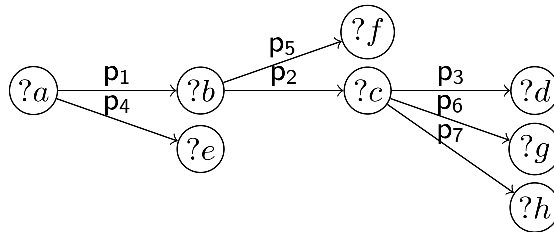


Fig. 7.2. Query Structure Generation Example. First, triple patterns are combined as path-shaped joins, i. e. ($?a$, p_1, $?b$), ($?b$, p_2, $?c$), ($?c$, p_3, $?d$), then star-shaped joins are added.

The main challenge for the query generation is to ensure the cardinality constraints, i. e. all queries should involve the required number of data sources and return non-empty result sets. Given that there exists at least one solution for creating a query according to a specified parameterization, then the naive approach would work as follows. First, create the specified query graph pattern by using triple patterns which contain only variables. Evaluate a SPARQL `SELECT` query with this graph pattern on the actual data sources to obtain all possible solutions for the variable bindings. Determine the number of involved data sources and the cardinality for each result set. Remove all solutions for variable bindings which do not satisfy the specified query constraints. Pick a random subset of the remaining solution sets and create the corresponding queries with bound RDF terms.

In reality, this approach is infeasible for the Linked Data cloud, because an efficient large scale federation implementation is not available. The distributed query evaluation of a large number of queries with many joins will simply take too long. Therefore, the SPLODGE query generator employs an iterative approach based on statistical information and cardinality estimation to create path-join and star-join patterns and ensure all cardinality constraints. Starting with one triple pattern, the query graph structure is extended stepwise by

adding one more triple pattern in each iteration until the complete query structure has been created. The problem, however, is to pick a triple pattern (with bound predicate) such that the extended query satisfies all constraints.

The query construction process starts with combining path-join triple patterns before adding star-join triple patterns to the query (cf. Def. 7.1). This order has been chosen because the number of possible path-joins across multiple data sources is essentially much smaller than the number of star-join patterns which are satisfiable within one data source. Therefore, it is the most restrictive part of the query parameterization.

Definition 7.1. *Path-Joins and Star-Joins are basic graph pattern with a set of associated data sources. Let* $\mathcal{F} = \{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$ *be a federated graph and let*

$$\mathcal{P} = (\mathcal{T}_1, \ldots, \mathcal{T}_n) \in ((U \cup V) \times (U \cup V) \times (U \cup L \cup V))^n, \ n \in \mathbb{N}$$

be a basic graph pattern. Then, a Path-Join is a "sequence" of triple patterns across data sources such that

$$PathJoin_{\mathcal{F}}(\mathcal{P}) : \forall\ 0 < i < n: \ obj(\mathcal{T}_i) = subj(\mathcal{T}_{i+1}) \ \wedge \ \exists\ j : eval(\mathcal{T}_i, \mathcal{G}_j) \neq \emptyset$$

The Star-Join defines resource-centric triple patterns which are matched as a whole by different data sources such that

$$StarJoin_{\mathcal{F}}(\mathcal{P}) : subj(\mathcal{T}_1) = \ldots = subj(\mathcal{T}_n) \ \vee \ obj(\mathcal{T}_1) = \ldots = obj(\mathcal{T}_n) \ \ and$$
$$\exists\ \mathcal{G} \in \mathcal{F}: \ \forall\ 0 < i \leq n: \quad eval(t_i, \mathcal{G}) \neq \emptyset$$

Path-Join Construction

According to the parameterization Path Joins have to include n triple patterns and span m data sources (with $m \leq n$). A path (i. e. link) across two data sources is defined by two RDF triples which are located in different data sources but contain the same URI in subject and object position, respectively. It is not sufficient to have just one RDF triple which links to resource URI from a different namespace. The second RDF triple is required such that the actual instantiation of the resource with respective properties can be matched by a query. Due to the requirement that each triple pattern in a path-join has to have a bound predicate the query generation relies on *predicate path pairs.*

Definition 7.2 (Predicate Path Pair). *A pair of triple patterns* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *with bound predicates* p_1 *and* p_2 *which match connected RDF triples across data sources* $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{F}$ *are referred to as predicate path pair.*

$$\mathcal{PPP}(\mathcal{T}_1, \mathcal{T}_2) = (p_1, \mathcal{G}_1, p_2, \mathcal{G}_2) \quad if\ eval(\mathcal{T}_1, \mathcal{G}_1) \neq \emptyset \wedge eval(\mathcal{T}_2, \mathcal{G}_2) \neq \emptyset$$

The set of all predicate path pairs in $\mathcal{F}$ *is defined as*

$$\mathcal{L}_{\mathcal{F}} = \{(p_1, \mathcal{G}_1, p_2, \mathcal{G}_2) \mid \exists\ s, o, x \in U \cup B \cup L: \ (s, p_1, x) \in \mathcal{G}_1 \wedge (x, p_2, o) \in \mathcal{G}_2\}$$

The creation of path-joins which span two or more data sources basically relies on the combination of multiple predicate path pairs, e. g. a path-join with three triple patterns across three data sources is defined by the join of $\mathcal{PPP}(\mathcal{T}_1, \mathcal{T}_2) = (p_1, \mathcal{G}_1, p_2, \mathcal{G}_2)$ and $\mathcal{PPP}(\mathcal{T}_2, \mathcal{T}_3) = (p_2, \mathcal{G}_2, p_3, \mathcal{G}_3)$. Example 7.3 illustrates this approach.

Example 7.3. A combination of two predicate path pairs which yields a path-join with three triple patterns.

$$((?s, \text{dc:creator}, ?x),(?x, \text{owl:sameAs}, ?o)) \textit{ and } \\ ((?s, \text{owl:sameAs}, ?y), (?y, \text{dbpp:name}, ?o))$$

Note that the triple pattern $\mathcal{T}_2$ must be the same in both predicate path pairs, i. e. have the same predicate and the same data source association. Moreover, the individual result sets of $\mathcal{T}_2$, i. e. for the bindings of $(?x, ?o)$ and $(?s, ?y)$ respectively, must overlap in order to return a result for the resulting path-join. The functions ϕ and τ define which values can be matched with the first or second pattern in a predicate path pair.

$$\phi(p_1, \mathcal{G}_1, p_2, \mathcal{G}_2) = \{(s, x) \mid (s, p_1, x) \in \mathcal{G}_1 \, \wedge \, (x, p_2, o) \in \mathcal{G}_2\}$$
$$\tau(p_1, \mathcal{G}_1, p_2, \mathcal{G}_2) = \{(x, o) \mid (s, p_1, x) \in \mathcal{G}_1 \, \wedge \, (x, p_2, o) \in \mathcal{G}_2\}$$

Hence, the condition for ensuring non-empty result for the combination of two predicate path pairs is

$$\tau(p_1, \mathcal{G}_1, p_2, \mathcal{G}_2) \cap \phi(p_2, \mathcal{G}_2, p_3, \mathcal{G}_3) \neq \emptyset$$

Path-Join Cardinality

The cardinality estimation for path-joins is similar to the cardinality estimation of equi-joins in relational databases (cf. Chapter 5). It requires knowledge about the cardinality and the join selectivity of two triple patterns. Such information is provided through statistics about the occurrence and correlation of predicates in triple patterns per data source.

Definition 7.4 (Path-Join Cardinality). *Let a path-join be defined by a set of predicate path pairs where $(p_1, \mathcal{G}_1), \ldots, (p_n, \mathcal{G}_n)$ define n triple patterns with bound predicate p_i and associated data source G_i. Then the Path-Join cardinality is the product of the individual triple pattern cardinalities $\sigma_{p_i}(\mathcal{G}_i)$ and the respective join selectivity for all matching predicate path pair tuples.*

$$|PathJoin_{\mathcal{F}}(\mathcal{P})| = \prod_{i=1..n} |\sigma_{p_i}(\mathcal{G}_i)| \; \cdot \prod_{i=2..n\text{-}1} sel(\mathcal{PPP}(\mathcal{T}_{i-1}, \mathcal{T}_i) \bowtie \mathcal{PPP}(\mathcal{T}_i, \mathcal{T}_{i+1}))$$

with $\sigma_{p_i}(\mathcal{G}_i) = \{(s, p_i, o) \in \mathcal{G}_i\}$ *and*

$$sel(\mathcal{PPP}(\mathcal{T}_{i-1},\mathcal{T}_i) \bowtie \mathcal{PPP}(\mathcal{T}_i,\mathcal{T}_{i+1})) = \frac{|\tau(p_{i-1}, \mathcal{G}_{i-1}, p_i, \mathcal{G}_i)| \cdot |\phi(p_i, \mathcal{G}_i, p_{i+1}, \mathcal{G}_{i+1})|}{|\sigma_{p_i}(\mathcal{G}_i)|^2}$$

Star-Join Construction

The parameterization of a Star-Join defines the number of triple patterns to be included, the position of the join variable for all triple patterns, i. e. subject or object position, and the number of data sources to be involved for returning matching resources. Moreover, a combination with another join pattern can be specified with an *extension point* in order to build complex queries. In the latter case the triple pattern of the extension point is automatically included in the Star-Join, e. g. triple pattern $(?a, \mathsf{p}_1, ?b)$ in Fig. 7.2 is part of a Path-Join and a connected Star-Join.

The construction of Star-Joins also relies on triple patterns with bound predicates. Again, an evaluation of a Star-Join pattern with unbound variables on the actual data sources, in order to find suitable predicate bindings, is too expensive and statistical data will be used as well. But in contrast to Path-Joins, which employ statistics about links between resources, the Star-Join construction requires only resource-centric information which can be easily maintained with *Characteristic Sets* [171].

Definition 7.5 (Characteristic Sets). *Resources in RDF graphs can be characterized by their associated predicates. Thus, the characteristic set of a resource s in data source $\mathcal{G} \in \mathcal{F}$ is basically the set of all its predicates.*

$$\mathcal{cs}_{\mathcal{G}}(s) := \{\, p \mid \exists\ o :\ (s, p, o) \in \mathcal{G}\}$$

Further, the set of all characteristic sets in $\mathcal{F}$ is

$$\mathcal{cs}_{\mathcal{F}} := \{\, \mathcal{cs}_{\mathcal{G}}(s) \mid \forall\, \mathcal{G} \in \mathcal{F} :\ \exists\ p, o :\ (s, p, o) \in \mathcal{G}\}$$

Analogously, inverse characteristic sets *are defined for the co-occurrence of predicates in the incoming links of a resource o in data source $\mathcal{G} \in \mathcal{F}$.*

$$\overline{\mathcal{cs}_{\mathcal{G}}}(o) := \{\, p \mid \exists\ s :\ (s, p, o) \in \mathcal{G}\}$$

Additionally, all inverse characteristic sets in $\mathcal{F}$ are defined by

$$\overline{\mathcal{cs}_{\mathcal{F}}} := \{\, \overline{\mathcal{cs}_{\mathcal{G}}}(o) \mid \forall\, \mathcal{G} \in \mathcal{F} :\ \exists\ s, p :\ (s, p, o) \in \mathcal{G}\}$$

Characteristic Sets are essentially equivalence classes for resources. Thus, each resource can only be part of one Characteristic Set. The original definition of Characteristic Sets [171] includes the number of resources and the frequency of each predicate in order to maintain information about multi-valued resource attributes. SPLODGE extends Characteristic Sets with a list of relevant data sources and the respective number of resources and the predicate frequency per data source. During query construction Characteristic Sets are employed for finding suitable combinations of predicates such that the Star-Join constraints can be satisfied as well as to determine the respective result cardinality and the number of involved data sources.

The Path-Join construction uses an iterative approach to extend the set of triple patterns incrementally. This is not necessary for Star-Joins because all possible combinations of predicates in Star-Joins are already represented in the Characteristic Sets. Hence, the Star-Join construction works as follows. For a Star-Join with n triple patterns and m resources, a subset of Characteristic Sets is selected such that the number of covered predicates and data sources is greater or equal to n and m, respectively. If a Star-Join has a predefined triple pattern, all Characteristic Sets which do not contain the respective predicate and possibly associated data source will be excluded. Moreover, all Characteristic Sets with less than n predicates will be ignored. From the remaining Characteristic Sets a random combination of n predicates will be selected.

Star-Join Cardinality

In order to determine the results size of a Star-Join it is only necessary to determine the supersets of the characteristic set of the joined triple patterns. This can be done based on the statistical information in Characteristic sets.

Definition 7.6 (Star-Join Pattern Cardinality). *The results set of a star-join will contain resource s iff the star-join's characteristic set $\mathcal{CS}(\mathcal{P})$ is a subset of the characteristic set $\mathcal{CS}_{\mathcal{G}}(s)$.*

$$|StarJoin_{\mathcal{F}}(\mathcal{P})| \;=\; |\{s \,|\, \forall\, \mathcal{G} \in \mathcal{F}:\; \mathcal{CS}(\mathcal{P}) \subseteq \mathcal{CS}_{\mathcal{G}}(s)\}|$$

The resulting number of resources is exact. Hence, the cardinality for star-join patterns can always be computed without estimation errors.

Combination of Join Patterns

The combination of two join patterns is done by defining an *extension point*, i. e. a triple pattern, in a previously created join pattern, and adding a new join pattern which uses the extension point as its initial triple pattern. Some restrictions apply for the definition of extension points. First, only Path-Joins are used to define extension points. However, they can be chosen randomly to allow for variations in the structure of the resulting queries while ensuring similar join patterns. Second, it is important to define the position of the join variable in an extension point, i. e. if it should be located in subject or object position. This is especially important for extensions with Path-Joins. A join via subject variable is done with the first triple pattern of the extending Path-Join while a join via object variable is done with the last triple pattern. Join patterns will be combined via `UNION` if no extension point is specified.

7.3.3 Validation of Generated Queries

The query generator should only return queries which satisfy all constraints specified in the query parameters. A query's structural constraints are already covered by the iterative query building approach. However, the cardinality constraints are not so easy to validate because executing a query and then counting the number of results is typically prohibitively expensive. The main reasons are the large size of the employed Linked Data sources (which need to be accessible through a federation or a large data warehouse) and the intentionally high complexity of certain queries. Hence, the cardinality validation has to rely on the statistical data and heuristics to estimate the likelihood that a query will actually return results. A naïve heuristic may exclude queries with an estimated result cardinality smaller than a certain threshold (e. g. 1 or any other suitable value). Unfortunately, a query's estimated cardinality is often inaccurate and can differ significantly from the actual result size. Especially queries with many joins are affected because the estimation error grows with every join as the estimated pattern selectivities are multiplied. Therefore, the validation relies on a specific confidence value which is computed for each query. It indicates the likelihood that a query satisfies the cardinality constraints, primarily that a query will return non-empty result sets. SPLODGE defines the confidence value based on the individual join selectivities in a query graph pattern. This allows for rejecting queries with very selective join combinations which are likely to produce no results. The evaluation of the SPLODGE approach in the next section will show the influence of this confidence value on the query result sizes.

7.4 Evaluation

The purpose of SPLODGE is to provide a methodology and tool set for the generation of benchmark queries across federated Linked Data sources. Hence, it does not define which set of queries should be included in a benchmark suite. This problem has been investigated by Montoya et al. [164]. Consequently, following evaluation is not a SPLODGE-based comparison of different federation systems. Instead, it investigates if the SPLODGE query generator is able to produce suitable benchmark queries which respect to a given query parameterization.

First, a set of query parameterizations is defined which will be used to generate sets of 100 random queries using a selection of data sources from the Linked Data cloud. In order to study the effects of the aforementioned confidence value on the query generation the SPLODGE query generator is configured with different settings, i. e. a baseline and different confidence value thresholds. The baseline is a totally random selection of triple patterns. SPLODGE_{lite} does not use any confidence value whereas $\text{SPLODGE}_{10^{-4}}$, $\text{SPLODGE}_{10^{-4}}$, and $\text{SPLODGE}_{10^{-4}}$ employ a minimum join selectivity threshold of 0.0001, 0.001, and 0.01, respectively. The generated query sets are then

compared in order to assess their conformance to the defined query parameterizations. Especially, the query result cardinality is in the focus of the evaluation since its estimation is the most challenging part of the query generation. Therefore, each query is executed on the actual data. Then the different query sets are compared with respect to the number queries with non-empty results and the accuracy of the estimation of the result cardinality.

7.4.1 Setup

The evaluation needs a suitable subset of real Linked Data sources from the Linked Data cloud and respective statistical information as required by the query generator. Moreover, the evaluation requires query parameterizations which are suitable for testing the main design goal of the SPLODGE methodology, i. e. the capability to create random queries across multiple data sources.

Dataset

There are numerous datasets in the Linked Data cloud, with different sizes, various schemas, and varying characteristics such as their degree of structuredness [70]. A large number of these datasets can be found on the CKAN[1] *Data Hub*[2] which is maintained by the *Open Knowledge Foundation*[3]. Live queries on SPARQL endpoints are infeasible for a federation benchmark because the performance suffers from the network delay and the results are not reproducible. Therefore, a snapshot of the datasets is required which can be obtained from data dumps or by crawling Linked Data resources. However, links between datasets in the Linked Data cloud are usually sparse [192] and the selection of a suitable subset, with enough connections such that a sufficient large number of queries across multiple data sources can be created for the evaluation, is not straightforward. Hence, the evaluation of SPLODGE uses crawled data from the *Billion Triple Challenge*.

The *Billion Triple Challenge* (in short BTC) is an annual competition in the context of the *International Semantic Web Conference* (ISWC). Its purpose is to encourage the development of useful applications for very large real life semantic web datasets. Each year a new dataset is provided which is the result of a crawl on the Linked Data cloud. Hence, it is an authentic snapshot of actual data found in the Linked Data cloud and it is often used by researchers beyond the scope of the Billion Triple Challenge. Authentic data also means that the BTC dataset is of varying quality and that it contains syntactic and semantic errors. Table 7.2 gives an overview of the characteristics of the Billion Triple Challenge datasets from past years. The SPLODGE evaluation uses the BTC dataset of 2011, which has the highest number of unique triples.

[1] http://ckan.org/
[2] http://thedatahub.org/
[3] http://okfn.org/

Table 7.2. Statistics of different Billion Triple Challenge Datasets

	BTC 2010	BTC 2011	BTC 2012
Total Size	624 GB	450 GB	303 GB
Total Quads	3.171.793.030	2.178.395.469	1.436.545.545
Unique Quads	3.154.896.097	2.145.122.248	1.311.765.894
Unique Triples	1.426.831.520	1.968.347.976	1.056.184.911
Contexts/Documents	8.132.721	7.423.477	9.283.829
Common Domains	22.299	789	837
Types	168.482	314.448	296.607
Predicates	95.589	47.738	57.257

The evaluation of benchmark queries on the BTC datasets is challenging, because an efficient implementation of a federation system for the Linked Data cloud does not exist yet. However, the measurement of the result cardinalities of queries does not need a distributed setup of the BTC data as it is independent of the actual data distribution. Thus, a data warehouse approach is used to store all data in a local repository with optimized indexes for fast query execution. But the size of the BTC dataset is even challenging for a centralized triple store implementation. Therefore, the SPLODGE evaluation utilizes RDF3X [172], a highly optimized and fast triple store implementation which can deal with very large RDF datasets. The storage of the 2 billion unique RDF triples of the 2011 BTC dataset in RDF3X with all indexes requires 150 GB of disk space. Most queries can be answered by RDF3X within seconds. But the evaluation of certain queries with large intermediate result sets can take up to several hours. Therefore, a timeout of two minutes is applied for each query.

Data Preprocessing

The evaluation setup includes the generation of all statistical information required by the SPLODGE query generator, i. e. the predicate path index and the characteristic sets. Analyzing the co-occurrence of predicates in path-shaped and star-shaped graph patterns in the benchmark dataset is a complex and expensive task which needs to be done in an off-line pre-processing step. The general problem is that the complete dataset cannot be kept in main memory for the analysis but needs to be processed in chunks. Thus, the relevant RDF triple for certain predicate combinations may be located in different chunks. Sorting can improve the data processing but is it also expensive and basically needs to be done for subjects and objects. Therefore, all statistics need to be extracted in an efficient way such that the memory usage, the amount of data stored on disk, and the overall processing time is minimized.

SPLODGE implements a pre-processing pipeline with different stages (cf. Fig. 7.3) that reads RDF data directly from chunked input files in the *NQuads* format, i. e. each text line contains (subject, predicate, object, context) tuples. The RDF data may contain syntactic errors and semantic inconsistencies, but SPLODGE does not apply any validation or correction mechanism. Following is a detailed description of the operations performed in each pre-processing step.

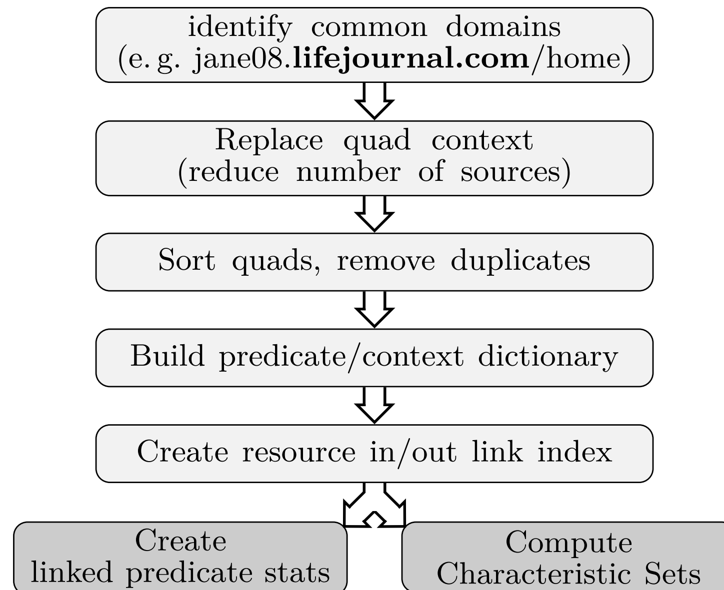


Fig. 7.3. Pre-processing steps for generating the datasets statistics

Identify Common Domains The Billion Triple Challenge dataset contains contexts information (document URIs) to refer to the original sources the data comes from. There are many contexts which belong to the same domain, e. g. livejournal.com has different sub-domains for each user, like {john,jane}.livejournal.com. SPLODGE identifies *common domains* through context URIs which have at least two other contexts as direct sub-domains.

Merging Quad Contexts SPLODGE is based on federated data source. Hence, the many different context URIs in the BTC data, which often contain just a few RDF triples, are not suitable for the query generation. Hence, an aggregation is done which replaces all context URIs with their respective common domain. This basically reduces the number of distinct contexts in the 2011 BTC data from several million document URIs to roughly 800 common domains. (cf. Tab. 7.2).

Sorting and Duplicate Removal Duplicate triples (or quads) are typically encountered in the BTC datasets. However, the statistics require frequency counts for the distinct number of elements. Hence, sorting and duplicate removal has to be done. Moreover, the generation of predicate path statistics benefits from sorted input data because it requires a group-

ing of RDF triples by the same resources in subject and object position. But sorting the complete dataset twice, i. e. by subject and by object, is expensive with respect to sorting time and the required disk space. Therefore, it will only be done for the subject and the subsequent processing step has to aggregate objects separately. Finally, an external sort mechanism is required since sorting of such a large dataset as cannot be done in memory, In fact, the standard UNIX `sort` utility is used as it already provides an efficient implementation of an external sort algorithm for large files including support for duplicate removal.

Building Dictionaries Predicate URIs and context URIs are quite verbose. But an efficient storage of indexes and statistics requires a compact data representation. In fact, the goal is to keep all data in main memory in order to allow for fast processing of the maintained statistical data. Hence, the next pre-processing step creates two dictionaries, i. e. for predicates and for common domains, which associate each URI with an integer value. This is a common approach for reducing the size of indexes. Moreover, knowledge about the actual URIs is not required for the following pre-processing steps and operations on integer values can be done much faster.

Create Resource Predicate Index The query generator requires statistical information about the co-occurrence of predicates. Therefore, all pairs of triple patterns which are connected by the same resource, either in a path-shaped or star-shaped join combination, have to be determined. Due to the large number of resources in the Linked Data cloud, i. e. several hundred thousand URIs for the 2011 BTC dataset (cf. Table 7.3), the Resource Predicate Index cannot be build in memory. Therefore, the BTC data is processed in chunks. In the first step a Resource Predicate Index is created for each chunk. Thereafter, they are sorted and merged.

Table 7.3. Number of Entities and Blank Nodes in the 2011 BTC Dataset

	Entities	BlankNodes
Total	485.630.664	382.608.765
as Subject	406.183.064	382.489.863
as Object	470.731.674	374.280.894
as Subject & Object	391.284.074	374.161.992

Each entry of the Resource Predicate Index describes one resource, i. e. it contains all predicate/context combinations for the resource's in-links $\mathcal{P}_{in}$ and out-links $\mathcal{P}_{out}$ with respective frequency counts.

$$\mathcal{P}_{in} = \{(p_i, \mathcal{G}_i, |eval_{\mathcal{G}_i}(s, p_i, r)|)\}$$
$$\mathcal{P}_{out} = \{(p_j, \mathcal{G}_j, |eval_{\mathcal{G}_j}(r, p_j, o)|)\}$$

Table 7.4 shows an example of the information stored in the Resource Predicate Index. It has three columns. The first contains all resources while the second and third store all information about in-links and out-links, respectively. Based on this index structure the creation of the Predicate Path Index and the Characteristic Set Index requires only a single pass over the Resource Predicate Index. Thus pre-processing cost and memory consumption is minimized.

Table 7.4. The Resource Predicate Index contains tuples of resources, in-links $\mathcal{P}_{in}$, and out-links $\mathcal{P}_{out}$. Each link entry is a set of tuples containing information about predicate, data source, and RDF triple count.

r	$\mathcal{P}_{in}$	$\mathcal{P}_{out}$
http://0-17.livejournal.com/	{(foaf:openid, livejournal.com, 1), (foaf:weblog, livejournal.com, 3)}	{(lj:dateCreated, livejournal.com, 1)}
http://aladiw.us	{(foaf:homepage, identi.ca, 2), (foaf:homepage, status.net, 1)}	{}
⋮	⋮	⋮

Create Path and Star Pattern Statistics The final step generates predicate path statistics and characteristic sets from the information in the Resource Predicate Index. Predicate Path statistics are essentially obtained by the Cartesian product of the predicates from all in-links with the predicates of the out-links of all resources in the Resource Predicate Index. Resources which have only in-links or only out-links are ignored because they are not part of a predicate path. Table 7.5 shows example statistics for Predicate Path statistics.

Table 7.5. Predicate Path Statistics. Each row defines a triple path $(\mathcal{T}_{p_1}, \mathcal{T}_{p_2})$ with predicates p_1 and p_2 such that $\mathcal{T}_{p_i} = \{(s, p_i, o) \in \mathcal{G}_i\}\ \forall\ s \in U, o \in U, \mathcal{G}_i \in \mathcal{F}$. Predicate and graph URIs are replaced with integer values in the actual index file.

$p_1 \rightarrow p_2$	$\mathcal{G}_1$	$\|\mathcal{T}_{p_1}\|$	$\mathcal{G}_2$	$\|\mathcal{T}_{p_2}\|$
owl:sameAs → foaf:knows	http://data.gov.uk	: 22	http://dbpedia.org	: 31
owl:sameAs → foaf:knows	http://open.ac.uk	: 58	http://dbpedia.org	: 17
rdfs:seeAlso → rdf:type	http://bio2rdf.org	: 15	http://www.uniprot.org	: 38
rdfs:seeAlso → rdfs:label	http://zitgist.com	: 49	http://musicbrainz.org	: 36
⋮	⋮		⋮	

Characteristic sets are based on the co-occurrence of predicates in the out-links of a resource. Inverse characteristic sets use the in-links instead. Table 7.6 shows example data stored in a characteristic set. The index entries for each combination of predicates and relevant data sources with matching resources includes the number of resources and the number of triple patterns for each predicate in the characteristic set. Due to multi-valued attributes the triple count can be higher than the number of resources.

Table 7.6. Characteristic Set Statistics. Predicate and graph URIs are replaced with integer values in the actual index file.

$\overline{p} = \{p_1, p_2, p_3, \ldots\}$	$\mathcal{G}$	$\lvert R_{\overline{p}:\mathcal{G}}\rvert$	$\lvert \mathcal{T}_{p1}\rvert$	$\lvert \mathcal{T}_{p2}\rvert$	$\lvert \mathcal{T}_{p3}\rvert$
rdf:type, rdfs:label, owl:sameAs	http://bio2rdf.org	632	632	844	632
—"—	http://www.uniprot.org	924	924	924	924
—"—	http://data.gov.uk	1173	1421	1173	1399
rdf:type, rdfs:label, foaf:name	http://dbpedia.org	3981	3981	5536	3981
⋮	⋮	⋮	⋮	⋮	⋮

All processing steps have been implemented with Shell scripts and Perl scripts which are optimized for low memory consumption and fast processing speed. Thus, they can also be run on typical commodity hardware. The source code of the scripts and of the query generator have been published as open source[4] under the LGPL license.

Query Parameterization

The goal of the evaluation is to verify that the SPLODGE query generator can produce useful distributed benchmark queries, i. e. queries with graph patterns that match connected resources across multiple data sources. It is one of the major contributions that SPLODGE can automatically generate such queries. Therefore, the parameterization of the evaluation queries defines path-joins with 3-6 triple patterns. Each triple pattern must be evaluated at a different data source. The generation of such queries is especially challenging because such triple pattern path are rare. The query generator produces sets of 100 random queries for the four different query types, i. e. path-joins with 3-6 triple patterns. The seed for the random number generator is fixed which allows to make the query generation reproducible.

[4] http://code.google.com/p/splodge/

7.4.2 Results

The evaluation results are presented according to three different measured aspects, i. e. the ratio of queries with non-empty results in a query batch, the distribution of result cardinalities in a query batch, and the comparison of the estimated result size with the real result size.

Result Cardinality

The investigation of the query result cardinality is divided into a quantitative evaluation which measures the result size of each generated query and a qualitative analysis of the compliance of estimated and real result sizes. First, in each batch of 100 path-join queries with three to six triple patterns, the number of queries which return non-empty results is counted. This is the primary indicator for the ability of the query generator to return queries which satisfy the result cardinality constraint of the query parameterization. Additionally, the distribution of result sizes is investigated.

Figure 7.4 shows the number of generated queries with non-empty results for each query generator configuration, comparing the baseline with the SPLODGE approach using no confidence value (SPLODGE*lite*) and increasing confidence values ($\mathsf{SPLODGE}_x$), where x defines a minimum join selectivity of 0.0001, 0.001, or 0.01, respectively.

First of all, the baseline algorithm, which uses only random triple patterns for the query construction, fails to create a single path-join query pattern that can be matched on the BTC dataset and return at least one result. Next, the SPLODGE*lite* approach can only produce 22 queries with non-empty result set for three triple patterns and significantly less for longer path-joins with just one query for the batch of queries with six triple patterns. SPLODGE with confidence values is able to produce considerably more queries with non-empty result sets. The number goes up to around 60% for three triple patterns whereas the join selectivity threshold does not make much of a difference. For queries with four to six path-join triple patterns there is an obvious drop in the number of non-empty results. This drop is more obvious for a less strict join selectivity threshold of 0.001 or 0.001. However, the highest threshold of 0.01 prevents the creation of any path-join query with six triple patterns, because there basically exist no triple pattern combination which satisfies all query parameters and additionally has this minimum join selectivity.

The variation of the result sizes is shown in the second plot of Fig. 7.4. It depicts minimum and maximum, as well as 0.2, 0.5 (median), and 0.8 quantiles for all non-empty result sizes in a query batch. For most query batches the minimum results size is 1 and the median lies between 10 and 100. A larger variation is observed for query batches with just a few non-empty query results. Query batches with a higher number of non-empty result sets are more similar. This similarity is most obvious, even including the 0.2 and 0.8 quantiles, for SPLODGE query batches with three or five triple patterns.

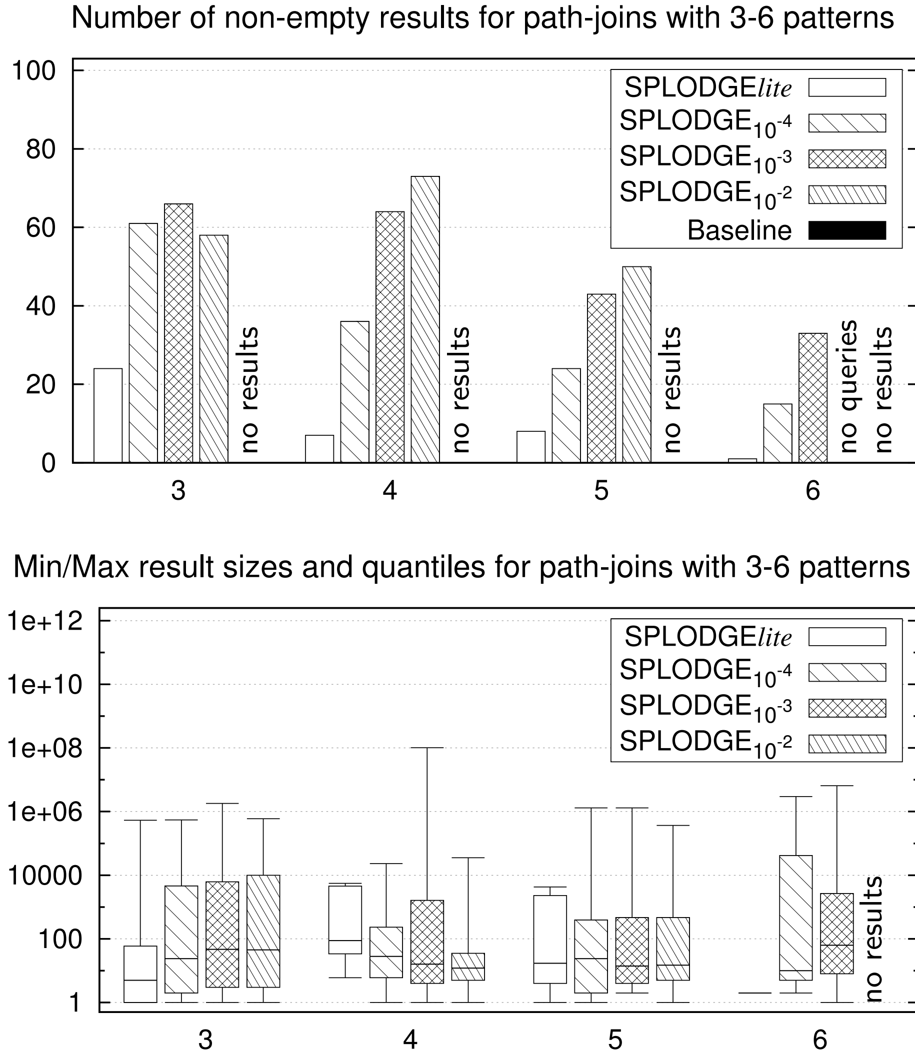


Fig. 7.4. Comparison of SPLODGE*lite* and SPLODGE using different confidence values, i. e. minimum join selectivity of 0.0001, 0.001, or 0.01, respectively. For each batch of 100 path-join queries with 3-6 triple patterns the number of non-empty results are compared (above). Moreover, the minimum and maximum result size as well as the quantiles for 0.2, 0.5 and 0.8 are shown for each query batch (below).

The maximum size of the result cardinality is around 1 million for several query batches with some outliers below and beyond.

The large spread of the observed result cardinalities, especially for path-joins with four triple patterns, which ranges from 1 and to several million, needs to be controlled by the query generator in order to produce queries which satisfy the result cardinality constraint as well. Therefore it is necessary to investigate how the estimated result size correlates to the real result size. Figure 7.5 shows a comparison of both values for the path-join queries with four triple patterns. The general observation is that there are only a few queries where estimated result size is close to the real result size (data points near the line). In fact, the result cardinality is overestimated in most cases (data points above the line) and there is also a considerable number of queries

with underestimated result sizes (data points below the line). For many cases this mis-estimation is even off by by several orders of magnitude. With the use of minimum join selectivity as a confidence value it is possible to reduce the number of differences between estimated and real result size, i. e. the queries generated with the most strict threshold of 0.01 are generally closer to the line and have less extreme outliers. However, there is still a larger number of overestimated result sizes.

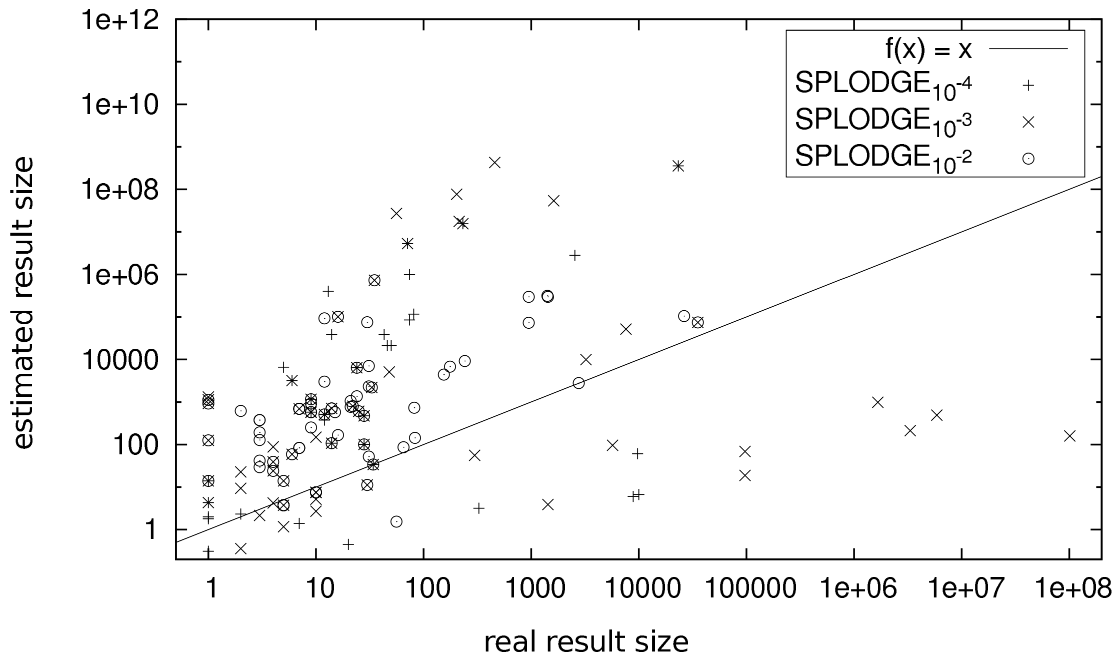


Fig. 7.5. Comparison of estimated and real result cardinality for path-join queries with four triple patterns. Data points above the line indicate overestimation, data points below the line indicate underestimation of the result cardinality.

The correct estimation of the result cardinality for SPARQL query patterns is generally challenging. In fact, estimation errors grow considerably with the number of joined triple patterns. While estimation errors are still acceptable to a certain degree for the query optimization (cf. Chapter 5), it is a problem for the query generator if a certain result cardinality is demanded by the query parameterization. However, it is not possible to improve the result cardinality estimation without including more statistical information about the graph structure of the datasets.

Query Analysis

The restriction of triple pattern combinations to certain graph structures and the use of cardinality constraints limit the number of RDF terms which can be chosen as bound predicates in the queries generated by SPLODGE. Therefore, the investigation of the frequency and distribution of predicate values may

reveal implicit characteristics of the queries. Table 7.7 shows the frequency of predicates for the different query generator settings. Obviously, many predicates come from the core Semantic Web vocabularies rdf, rdfs, and owl, e. g. rdf:type is the most prominent predicate and occurs in average in 1 out of 5 times in the queries. The rank of owl:equivalentClass is almost constant for the different algorithms while the ranking of the other predicates shows more fluctuation. Notable changes can be observed for skos:exactMatch and owl:sameAs, e. g. skos:exactMatch does not show up in the top 10 predicates except for $\text{SPLODGE}_{10^{-4}}$. However, for $\text{SPLODGE}_{10^{-2}}$ it is not used in any of the queries. Another interesting observation can be made for owl:sameAs which is usually considered as the standard predicate for defining links between datasets. However, as the threshold for the minimum selectivity increases it occurs less often in the queries. One reasonable explanation is that resources are typically not connected via a long chain of owl:sameAs relations across multiple datasets. Another interesting predicate is zemanta:targetType. Its popularity increases significantly with a higher threshold for the join selectivity. The reason is probably the reduction of the overall number of distinct predicates which qualify for being used in a query.

Table 7.7. Ranking of predicates occurring in the benchmark queries generated with SPLODGE *lite*, $\text{SPLODGE}_{10^{-4}}$, $\text{SPLODGE}_{10^{-3}}$, and $\text{SPLODGE}_{10^{-2}}$, respectively. The first column is the overall predicate frequency in all generated queries. The other columns indicate the individual predicate frequency and the according rank.

	all	SPLODGE*lite*	$\text{SPLODGE}_{10^{-4}}$	$\text{SPLODGE}_{10^{-3}}$	$\text{SPLODGE}_{10^{-2}}$
rdf:type	19.1 %	15.7 % (2)	18.6 % (1)	21.8 % (1)	21.0 % (1)
rdfs:seeAlso	8.4 %	16.3 % (1)	6.2 % (3)	5.3 % (4)	4.6 % (3)
owl:disjointWith	5.4 %	2.6 % (6)	3.7 % (5)	7.0 % (2)	9.5 % (2)
rdfs:subClassOf	3.9 %	3.1 % (5)	2.4 % (8)	6.5 % (3)	3.5 % (5)
rdfs:isDefinedBy	3.7 %	1.7 % (9)	6.5 % (2)	4.0 % (5)	1.9 % (10)
owl:equivalentClass	2.7 %	2.3 % (7)	2.7 % (7)	3.2 % (6)	2.7 % (8)
foaf:primaryTopic	2.4 %	3.9 % (4)	0.8 % (16)	1.9 % (7)	3.1 % (7)
rdfs:label	1.9 %	1.5 % (10)	1.6 % (10)	1.7 % (9)	3.2 % (6)
skos:exactMatch	1.8 %	1.3 % (11)	4.3 % (4)	0.9 % (18)	–
owl:sameAs	1.7 %	1.9 % (8)	3.2 % (6)	0.7 % (21)	0.8 % (14)
...					
zemanta:targetType	1.2 %	0.1 % (91)	0.5 % (29)	1.3 % (14)	3.8 % (4)
distinct predicates	687	402	353	283	191
total predicates	6600	1800	1800	1800	1200

7.5 Summary

Benchmarks are widely used for comparing the performance of different query processing implementations. However, there is a lack of Linked Data benchmarks for distributed SPARQL query processing. SPLODGE provides a novel methodology and a tool set for a systematic generation of SPARQL benchmark queries for federated Linked Data query processing systems. First, a thorough analysis of benchmark query characteristics was conducted, taking into account existing benchmarks and query characteristics of available query logs. Thus, three query parameter dimensions were defined, i. e. structural patterns, query complexity, and cardinality constraints. The resulting parameterization of benchmark queries is very flexible and can be used to define queries for common scenarios and also corner cases.

A core feature of SPLODGE is the automatic generation of random benchmark queries for a given query parameterization. Its general approach is an iterative join pattern construction with validation of cardinality constraints, i. e. especially result cardinality. This cannot be done on the actual data during query generation because it is prohibitively expensive. Instead, compressed dataset statistics for Path-Joins and Star-Joins are used. However, since estimation errors occur for the statistics-based cardinality computation the generated queries are only accepted as valid results if they have a high likelihood for returning non-empty results.

In order to verify that SPLODGE generates appropriate benchmark queries for a given query parameterization, an evaluation was conducted to access the "quality" of the generated queries. The evaluation was based on the 2011 Billion Triple Challenge dataset. All benchmark queries were executed on the dataset in order to measure their result size and to analyze how many queries could actually return results. A comparison with estimated result size revealed large difference between estimated and real result size. This is due to the correlations in the RDF data which can hardly be captured without detailed statistics. However, it was shown that SPLODGE is a flexible and scalable approach for automatic generation of useful SPARQL benchmark queries.

8

Conclusion

The scalability of information retrieval on top of the growing number of Linked Data sources has received a lot of attention in recent years. Especially, RDF data source federation is currently an active research area and it also has been the main research topic of this dissertation. Since the number of linked RDF datasets is increasing rapidly there is a need for systems which can provide easy access to this huge amount of data. However, this requires a scalable federation infrastructure and sophisticated algorithms for distributed query processing. Therefore, researchers have become quite interested in the adaption of well-known database approaches and query optimization techniques for Linked Data information retrieval because both research areas share similar challenges. But since there are also significant differences it is necessary to have a good knowledge of both domains in order to determine adequate combinations of appropriate solutions. So far, only few database researchers have been looking at Semantic Web technologies and on the other side the Semantic Web community is somehow reluctant to dive into 40 years of database research. Hence, there are still many open research questions.

This dissertation focused primarily on distributed query optimization for federated RDF data sources, including specific aspects like the requirements for a scalable Linked Data federation infrastructure, data source selection, and cost-based join order optimization. Moreover, solutions for distributed statistics management and benchmark query generation for Linked Data were presented. The main contributions of this thesis will be summarized in the next section. It is followed by a reflection of the lessons learned and a discussion of future research opportunities in this area.

8.1 Summary and Research Contributions

Due to the growing size of the Linked Data cloud, the number of different domains, and the diversity of the datasets, there are many new challenges concerning scalable and efficient data processing like in a web-scale database. One of the basic assumptions of this dissertation is that the federation of RDF data sources can benefit from the adoption of common techniques from distributed and federated databases which have been developed in four decades of research in that area. In order to provide a system for RDF data source federation a mediator-based architecture was chosen which implements a distributed query optimization strategy based on common database technologies, e. g. cost-based selection of optimal query execution plans. However, due to major differences between relational data and the RDF graph model there have been a number of related issues, like source selection, statistics management, and benchmarking, which had to be investigated as well. Therefore, this dissertation makes several contributions for building a flexible and scalable Linked Data federation infrastructures.

- The design and implementation of a mediator-based federation infrastructure for Linked Data. It is based on standard Semantic Web technology and incorporates common relational databases approaches in order to allow for efficient distributed SPARQL query processing.
- A source selection strategy which utilizes VOID descriptions for matching query patterns to data sources including a refinement with SPARQL ASK queries in order to split arbitrary SPARQL queries into suitable sub queries for a distributed execution.
- A join-order optimization approach for distributed SPARQL queries based on dynamic programming with a cost-based model that takes into account the specific characteristics of RDF graph data and employs VOID statistics for cardinality estimation in order to find the optimal query execution plan.
- An evaluation which compares SPLENDID with other state-of-the-art query optimization implementations and proofs that the use of the meta data and statistics from VOID descriptions allows for an efficient distributed query optimization.
- A query generation methodology for RDF federation benchmarks that scales with the number and sizes of the involved datasets and offers automatic generation of benchmark queries for more realistic evaluations on arbitrary datasets from the Linked Data cloud.

In addition, issues related to information retrieval on highly correlated data in a distributed setup have been studied that led to following results.

- A novel and scalable statistics management strategy for complex graph data with TF-IDF-based result ranking in a highly distributed system relying on a Peer-to-Peer network topology.

8.2 Research Obstacles and Lessons Learned

The initial motivation for developing SPLENDID was to provide efficient distributed SPARQL query execution for federated RDF data sources by optimizing the join order of triple patterns. But since different optimization criteria have to be considered it soon became apparent that simple heuristics are often not sufficient for effective join order optimization. Therefore, a more sophisticated query optimization strategy was needed and dynamic programming was chosen because it is a popular approach in relational databases and allows to find the best query execution plan. However, the overall query processing performance does not only depend on the query optimization. It is also influenced by the source selection, the efficiency of the query execution, and the accuracy of the employed data statistics. For example, even with a sophisticated query optimization strategy the distributed query execution can be slow if the source selection and the query execution are not implemented efficiently as well. Consequently, the research on an efficient Linked Data federation had to cover multiple aspects including source selection, query optimization, query execution, and suitable statistics which can be used for both source selection and query optimization.

The first prototype implemented the cost-based join order optimization on top of the Sesame RDF library. But it turned out that there is a strong dependency between the query optimization and the query execution, i. e. either a query optimizer can only employ the physical join operators which are supported by the query executer or, as in the case of SPLENDID, Sesame's query executor had to be extended in order to support an additional join algorithm. However, this dependency also makes a comparison with other federation implementations complicated because the components often cannot be tested in isolation. This very problem occurred when comparing SPLENDID with FedX and it required an in-depth analysis of the results. Although the evaluation was intended to compare the query optimization of both approaches it turned out that the differences in the results were caused by a better query execution in FedX. As a result SPLENDID's query execution adapted the FedX implementation. But this experience also leads to another important insight, i. e. the choice of employed join operators has a significant influence on the overall federator performance. Thus, instead of implementing a federator by following the distributed query processing workflow, i. e. source selection, query optimization, and query execution, it makes more sense to first find the most efficient join implementations for the investigated scenario, then focus on the query optimization, and finally deal with the source selection.

Dataset statistics are very important for the query optimization, especially for the cardinality estimation in SPLENDID's cost-based join order optimizations, i. e. the best query execution plans should be obtained when detailed statistics are used to compute accurate cost estimates. However, it is not possible to collect highly detailed statistics for an arbitrary number of Linked Data sources. Hence, it seemed quite challenging to produce good query ex-

ecution plans based on VOID statistics. But it turned out that SPLENDID would often produce the same query execution plan for the tested queries as FedX's heuristics-based approach. Moreover, the variation of the used statistics did have less effect on the execution plans for some queries than expected. Although this seemed surprising at first there is a simple explanation. The accuracy of the cardinality estimation always decreases with the number of joins due to growing estimation errors. Therefore, detailed statistics typically yield only better estimations for the first triple patterns and the first join. Since the benchmark included common queries with only a few joins both optimization approaches could generally determine the best combination of the "cheapest" SPARQL triple patterns. However, a cost-based query optimization can usually outperform heuristic for more complex queries and non-sequential query execution plans.

The need for a suitable data format to summarize dataset characteristics led to the use of VOID descriptions. Albeit VOID has become a de-facto standard it is not widely supported yet. Therefore, a significant pre-processing overhead is inevitable but it also allows to generate VOID description specifically tailored for SPLENDID's needs, i. e. for source selection and query optimization. As mentioned above even with limited statistical data it is possible to generate good query execution plans. On the other hand the use of VOID descriptions for the source selection turned out to be more problematic than initially expected. For instance, the list of potential data sources for SPARQL query patterns with properties and types from common vocabularies, like FOAF, can become quite large. But specific restrictions of the triple patterns, e. g. bound resource URIs or join combinations, can hardly be taken into account using only the information from the VOID descriptions. Therefore, the VOID-based source selection in SPLENDID had to be combined with a refinement mechanism, i. e. `ASK` queries, in order to reduce the number of requests to SPARQL endpoints.

Another problem throughout the work on SPLENDID was the lack of a suitable benchmark to test the system and compare it with other federation implementations. An adaption of existing centralized RDF benchmarks was not a viable option as the datasets and queries of these benchmarks could not resemble the characteristics of federated Linked Data scenarios. However, obtaining suitable federated SPARQL queries is problematic because there is still no live Linked Data federation system which could provide authentic query logs. Still, the only available benchmark is FedBench. But its limitation to a few datasets and the use of hand-picked queries does not allow for testing federated query execution on a larger scale within the Linked Data cloud. Thus, the results of a FedBench-based comparison between different federation implementations are only valid for restricted federation scenarios. But an automatic query generation for arbitrary datasets, as provided by SPLODGE is also challenging. In fact, the computation is very similar to the estimations of the query optimization, e. g. it requires sophisticated dataset statistics in order to estimate if a combination of triple pattern can likely produce some

results when evaluated on the datasets. The latter gets even harder with an increasing number of joins and more connected data sources. Additionally, the generated queries are not necessarily meaningful albeit they have the desired characteristics.

8.3 Outlook and Future Work

Although a lot of research has been done recently on different aspects of federated SPARQL query processing there is still no single system which provides a comprehensive solution for scalable Linked Data federation. The main reasons are the size of the Linked Data cloud, the diversity of the datasets, and – on the technical side – the dependencies between individual processing steps, like federation setup, source discovery, statistics management, source selection, query optimization, and query execution. In order to focus on specific federation aspects and also to simplify the implementation, all federation systems consider certain application scenario constraints, e. g. in the case of SPLENDID a static setup with pre-computed statistics and a query optimization strategy which follows the optimize-then-execute paradigm. A step-wise relaxation of these constraints would be among the next steps in order to allow for more flexible Linked Data federation, i. e. with dynamic data integration, and to scale up to a large number of diverse RDF data sources.

While designing and implementing SPLENDID's distributed query processing it became clear that there is a high potential for more research in the direction of adapting established database query optimization techniques for Linked Data query processing. In fact, there are two research aspect which seem to be most promising for further investigations, i. e. 1) using additional information about the graph structure and links between the RDF datasets for better source selection and cardinality estimation and 2) the application of adaptive query optimization strategies which allow for coping with unreliable network conditions and providing a responsive application which can return results even in case of failures of individual data sources. The first aspect is mainly about dealing with the high data correlation and dependencies in RDF graphs, which is often ignored for cardinality estimation due to the inherent complexity. Moreover, compressed data structures like histograms are commonly used in relational databases, but there is not yet any sophisticated adaption for RDF which can be incorporated in VOID descriptions and also allows for a differentiation of predicates, entities, and literals. The second aspect is also very important when consuming data from the Linked Data cloud. Although adaptive query processing has been intensively researched in the database area only few researchers have investigated an application for query processing on RDF data sources. Thus, especially for the larger number of small and often less reliable Linked Data sources this approach will be more beneficial than static query optimization.

Another area of improvement exists with respect to the setup of a Linked Data federation system. Instead of an explicit initial definition of a fixed set of data sources an automatic discovery mechanism should be able to add suitable RDF data sources to the federation on the fly. Therefore, information about datasets have to be obtained including their SPARQL endpoints and suitable VOID descriptions. Since the lookup of resource URIs (found in SPARQL queries) cannot provide such information it is necessary to rely on directory services like CKAN which *know* a large number of data sources and can answer requests that ask for data sources with certain vocabularies or containing resources with a specific data schema. Moreover, the gathering of the required VOID statistics has to be performed on demand. An expensive preprocessing step, as currently used by SPLENDID, would block the whole query processing if dynamic discovery of a new data sources is employed. Hence, sophisticated statistics collection and management strategies are required which would also benefit from a better support of VOID descriptions by the data providers or by third party directory services.

An effective statistics management in general is also a future research topic since Linked Data sources are dynamic, i. e. datasets get updated in unpredictable intervals. Hence, such updates need to be reflected in the index of a statistics-based federation system. There is still no standard for how such updates can be detected and propagated. At least on the technical side it seems reasonable to investigate update mechanism which can be combined with the provisioning of VOID descriptions.

Finally, scalable Linked Data federation systems face new challenges concerning their usability. Due to the abstraction of the underlying data sources a user may not know which vocabularies (ontologies) to use in order to formulate a SPARQL query that can return the desired results. Therefore, future federation systems need to guide the user in her search, e. g. by giving information about commonly used vocabularies and through auto completion features. On the other hand, the exact match semantics of SPARQL is problematic for returning all relevant results because different vocabularies may be used in the datasets to describe the same or similar resources. Query expansion is a common approach to solve this problem but it requires to collect and maintain information about mappings between the ontologies. As pointed out before, a broader support of VOID descriptions, e. g. with extensions to support such information would be beneficial for Linked Data federation systems.

References

1. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web data Management Using Vertical Partitioning. In: Proceedings of the 33rd International Conference on Very Large Data (VLDB), pp. 411–422. Vienna, Austria (2007)
2. Abdullah, H., Rinne, M., Törmä, S., Nuutila, E.: Efficient Matching of SPARQL Subscriptions Using Rete. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC ’12, pp. 372–377. ACM Press, Riva del Garda (Trento), Italy (2012)
3. Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R.: P-Grid: A Self-organizing Structured P2P System. ACM SIGMOD Record **32**(3), 29–33 (2003)
4. Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Pelt, T.: GridVine: Building Internet-Scale Semantic Overlay Networks. In: Proceedings of the 3rd International Semantic Web Conference (ISWC), pp. 107–121. Springer-Verlag, Hiroshima, Japan (2004)
5. Aberer, K., Datta, A., Hauswirth, M.: P-grid: Dynamics of self-organizing processes in structured peer-to-peer systems. In: R. Steinmetz, K. Wehrle (eds.) Peer-to-Peer Systems and Applications, LNCS, vol. 3485, chap. 10, pp. 137–153. Springer-Verlag (2005)
6. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: AN Adaptive query ProcesSing engIne for sparql enDpoints. In: Proceedings of the 10th International Conference on the Semantic Web, pp. 18–34. Springer (2011)
7. Adamku, G., Stuckenschmidt, H.: Implementation and Evaluation of a Distributed RDF Storage and Retrieval Systemb. In: Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI’05), pp. 393–396 (2005)
8. Akar, Z., Halaç, T.G., Ekinci, E.E., Dikenelli, O.: Querying the Web of Interlinked Datasets using VOID Descriptions. In: Proceedings of the Linked Data on the Web Workshop (2012)
9. Alexander, K., Cyganiak, R., Hausenblas, M., Zhao, J.: Describing Linked Datasets – On the Design and Usage of voiD, the “Vocabulary Of Interlinked Datasets”. In: Proceedings of the Linked Data on the Web Workshop. CEUR Workshop Proceedings, ISSN 1613-0073, Madrid, Spain (2009)
10. Androutsellis-Theotokis, S., Spinellis, D.: A Survey of Peer-to-Peer Content Distribution Technologies. ACM Computing Surveys **36**(4), 335–371 (2004)

11. Angles, R., Gutierrez, C.: Querying RDF Data from a Graph Database Perspective. In: Proceedings of the 2nd European Semantic Web Conference, c, pp. 346–360. Heraklion, Crete, Greece (2005)
12. Angles, R., Gutierrez, C.: Survey of Graph Database Models. ACM Computing Surveys **40**(1), 1–39 (2008)
13. Ansell, P.: Model and prototype for querying multiple linked scientific datasets. Future Generation Computer Systems **27**(3), 329–333 (2011)
14. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "Bit"loaded: A Scalable Lightweight Join Query Processor for RDF Data. In: Proceedings of the 19th International World Wide Web Conference, pp. 41–50. ACM Press, Raleigh, NC, USA (2010)
15. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A Nucleus for a Web of Open Data. In: Proceedings of the 6th International Semantic Web Conference, pp. 722–735. Busan, Korea (2007)
16. Avnur, R., Hellerstein, J.M.: Eddies: Continuously Adaptive Query Processing. In: Proceedings of the International Conference on Management of Data - SIGMOD'00, vol. 29, pp. 261–272. ACM Press, Dallas, TX, USA (2000)
17. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison Wesley (1999)
18. Bairoch, A., Apweiler, R., Wu, C.H., Barker, W.C., Boeckmann, B., Ferro, S., Gasteiger, E., Huang, H., Lopez, R., Magrane, M., Martin, M.J., Natale, D.A., O'Donovan, C., Redaschi, N., Yeh, L.S.L.: The Universal Protein Resource (UniProt). Nucleic Acids Research **33**(suppl 1), D154–D159 (2005)
19. Barbieri, D.F., Braga, D., Ceri, S., Grossniklaus, M.: An execution environment for C-SPARQL queries. In: Proceedings of the 13th International Conference on Extending Database Technology - EDBT '10, pp. 441–452. ACM Press, Lausanne, Switzerland (2010)
20. Barnaghi, P., Presser, M.: Publishing Linked Sensor Data. In: Proceedings of the 3rd International Workshop on Semantic Sensor Networks (SSN10). CEUR Workshop Proceedings, Shanghai, China (2010)
21. Bartolomeo, G.: A Spectrometry of Linked Data. In: Proceedings of the Workshop on Linked Data on the Web (2012)
22. Basca, C., Bernstein, A.: Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In: Proceedings of the 6th International Workshop on Scalable Semantic Web Knowledge Base Systems - SSWS'10, pp. 64–79. Shanghai, China (2010)
23. Basse, A., Gandon, F., Mirbel, I., Lo, M.: DFS-based frequent graph pattern extraction to characterize the content of RDF Triple Stores. In: Proceedings of the WebSci10: Extending the Frontiers of Society On-Line. Raleigh, NC, USA (2010)
24. Bawa, M., Garcia-Molina, H., Gionis, A., Motwani, R.: Estimating aggregates on a peer-to-peer network. Tech. rep., Computer Science Dept., Stanford University (2003)
25. Bayer, R., McCreight, E.: Organization and Maintenance of Large Ordered Indexes. pp. 173–189. Springer Verlag (1972)
26. Belleau, F., Nolin, M., Tourigny, N., Rigault, P., Morissette, J.: Bio2RDF: Towards a mashup to build bioinformatics knowledge systems. Journal of Biomedical Informatics **41**(5), 706–716 (2008)
27. Berners-Lee, T.: Linked Data Design Issues. http://www.w3.org/DesignIssues/LinkedData.html

28. Bernstein, P., Chiu, D.: Using Semi-Joins to Solve Relational Queries. Journal of the ACM **28**(1), 25–40 (1981)
29. Bernstein, P.A., Goodman, N., Wong, E., Reeve, C.L., Rothnie, J.B.: Query Processing in a System for Distributed Databases (SDD-1). ACM Transactions on Database Systems **6**(4), 602–625 (1981)
30. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. International Journal on Semantic Web and Information Systems **5**(3), 1–22 (2009)
31. Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia – A Crystallization Point for the Web of Data. Web Semantics: Science, Services and Agents on the World Wide Web **7**(3), 154–165 (2009)
32. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal on Semantic Web and Information Systems **5**(2), 1–24 (2009)
33. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
34. Böhm, C., Lorey, J., Naumann, F.: Creating voiD Descriptions for Web-scale Data. Web Semantics: Science, Services and Agents on the World Wide Web **9**(3), 339–345 (2011)
35. Bolles, A., Grawunder, M., Jacobi, J.: Streaming SPARQL - Extending SPARQL to Process Data Streams. In: S. Bechhofer, M. Hauswirth, J. Hoffmann, M. Koubarakis (eds.) Proceedings of the 5th European Semantic Web Conference, LNCS, vol. 5021, pp. 448–462. Springer-Verlag, Tenerife, Canary Islands, Spain (2008)
36. Brickley, D., Guha, R.: RDF Schema 1.1, W3C Recommendation 25 February 2014. http://www.w3.org/TR/rdf-schema/
37. Brickley, D., Miller, L.: FOAF Vocabulary Specification 0.99, Namespace Document 14 January 2014. http://xmlns.com/foaf/spec/
38. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics **1**(4), 485–509 (2004)
39. Broekstra, J., Kampman, A., Van Harmelen, F.: Sesame: A generic Architecture for Storing and Querying RDF and RDF Schema. In: I. Horrocks, J. Hendler (eds.) First International Semantic Web Conference - ISWC ’02, LNCS, vol. 2342, pp. 54–68. Springer-Verlag, Sardinia, Italy (2002)
40. Buil-Aranda, C., Arenas, M., Corcho, O.: Semantics and Optimization of the SPARQL 1.1 Federation Extension. In: G. Antoniou, M. Grobelnik, E. Simperl, et al. (eds.) 8th Extended Semantic Web Conference, pp. 1–15. Springer-Verlag, Heraklion, Crete, Greece (2011)
41. Cai, J., Poon, C.K.: Path-Hop: Efficiently Indexing Large Graphs for Reachability Queries. In: Proceedings of the 19th ACM International Conference on Information and Knowledge Management - CIKM ’10, pp. 119–128. ACM Press, Toronto, Ontario, Canada (2010)
42. Cai, M., Frank, M.: RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network. In: Proceedings of the 13th International World Wide Web Conference, pp. 650–657. ACM, New York, NY, USA (2004)
43. Cai, M., Frank, M., Chen, J., Szekely, P.: MAAN: A Multi-Attribute Addressable Network for Grid Information Services. Journal of Grid Computing **2**(1), 3–14 (2004)

44. Calbimonte, J.P., Jeung, H., Corcho, O., Aberer, K.: Semantic Sensor Data Search in a Large-Scale Federated Sensor Network. In: Workshop on Semantic Sensor Networks - SSN'11, pp. 14–29 (2011)
45. Carey, M.J., DeWitt, D.J., Naughton, J.F.: The 007 Benchmark. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 12–21. ACM Press (1993)
46. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs. Web Semantics: Science, Services and Agents on the World Wide Web **3**(4), 247–267 (2005)
47. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: Proceedings of the 14th International Conference on World Wide Web, pp. 613–622. ACM Press, New York, New York, USA (2005)
48. Carroll, J.J., Stickler, P.: TriX: RDF triples in XML. In: Proceedings of the Extreme Markup Languages Conference. Montréal, Quebec, Canada (2004)
49. Cattell, R.G.G.: The engineering database benchmark. In: The Benchmark Handbook for Database and Transaction Systems (2nd Edition). Morgan Kaufmann (1993)
50. Cattuto, C., Schmitz, C., Baldassarri, A., Servedio, V., Loreto, V., Hotho, A., Grahl, M., Stumme, G.: Network properties of folksonomies. AI Communications **20**(4), 245–262 (2007)
51. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems **26**(2), 1–26 (2008)
52. Chaudhuri, S., Dayal, U.: An Overview of Data Warehousing and OLAP Technology. ACM SIGMOD Record **26**(1), 65–74 (1997)
53. Cheng, G., Qu, Y.: Searching Linked Objects with Falcons: Approach, Implementation and Evaluation. International Journal on Semantic Web and Information Systems **5**(3), 49–70 (2009)
54. Cheung, K., Frost, H.R., Marshall, M.S., Prud'hommeaux, E., Samwald, M., Zhao, J., Paschke, A.: A journey to Semantic Web query federation in the life sciences. BMC Bioinformatics **10:S10** (2009)
55. Clark, K., Feigenbaum, L., Torres, E.: SPARQL Protocol for RDF, W3C Recommendation 15 January 2008. http://www.w3.org/TR/rdf-sparql-protocol/
56. Codd, E., Codd, S., Salley, C.: Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate. Tech. rep., Codd & Date, Inc (1993)
57. Crespo, A., Garcia-Molina, H.: Semantic Overlay Networks for P2P Systems. In: Agents and Peer-to-Peer Computing, pp. 1–13. Springer (2005)
58. Cudré-Mauroux, P., Agarwal, S., Aberer, K.: GridVine: An Infrastructure for Peer Information Management. IEEE Internet Computing **11**(5), 36–44 (2007)
59. Cyganiak, R., Reynolds, D.: The RDF Data Cube Vocabulary, W3C Recommendation 16 January 2014. http://www.w3.org/TR/vocab-data-cube/
60. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014. http://www.w3.org/TR/rdf11-concepts/
61. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM **51**(1), 107–113 (2008)
62. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of 21st ACM Symposium on

Operating Systems Principles - SOSP '07, vol. 41, pp. 205–220. ACM Press, Stevenson, WA, USA (2007)
63. Della Valle, E., Turati, A., Ghioni, A.: PAGE: A distributed infrastructure for fostering RDF-based interoperability. In: F. Eliassen, A. Montresor (eds.) DAIS'06 Proceedings of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems, vol. 4025, pp. 347–353. Springer-Verlag, Bologna, Italy (2006)
64. Deshpande, A., Hellerstein, J.M.: Lifting the burden of history from adaptive query processing. In: Proceedings of the 30th International Conference on Very Large Data Bases, pp. 948–959. ACM Press, Toronto, Ontario, Canada (2004)
65. Deshpande, A., Ives, Z., Raman, V.: Adaptive Query Processing. Foundations and Trends in Databases **1**(1), 1–140 (2007)
66. DeWitt, D.J.: The Wisconsin Benchmark: Past, Present, and Future. In: The Benchmark Handbook for Database and Transaction Systems (2nd Edition). Morgan Kaufmann (1993)
67. Ding, L., Finin, T., Joshi, A., Pan, R., Cost, R.S., Peng, Y., Reddivari, P., Doshi, V., Sachs, J.: Swoogle: A Semantic Web Search and Metadata Engine. In: Proceedings of the 13th ACM Conference on Information and Knowledge Management (CIKM), pp. 652–659. ACM Press, Washington DC, USA (2004)
68. Ding, L., Shinavier, J., Shangguan, Z., McGuinness, D.: SameAs Networks and Beyond: Analyzing Deployment Status and Implications of owl:sameAs in Linked Data. In: Proceedings of the 9th International Semantic Web Conference (ISWC), pp. 145–160. Springer (2010)
69. Dodds, L., Davis, I.: Linked Data Patterns. Creative Commons Attribution 2.0 (2012). http://patterns.dataincubator.org/book/
70. Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O.: Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In: Proceedings of the International Conference on Management of Data, pp. 145–156. ACM Press (2011)
71. Dürst, M.J., Suignard, M.: RFC 3987: Internationalized Resource Identifiers (IRIs). January 2005. http://www.ietf.org/rfc/rfc3987.txt
72. D'Aquin, M., Baldassarre, C., Gridinoc, L., Angeletou, S., Sabou, M., Motta, E.: Characterizing Knowledge on the Semantic Web with Watson. In: Proceedings of the 5th International Workshop on Evaluation of Ontologies and Ontology-based Tools (EON), pp. 1–10. Busan, Korea (2007)
73. Ebert, J., Franzke, A.: A Declarative Approach to Graph Based Modeling. In: Ernst W. Mayr, G. Schmidt, G. Tinhofer (eds.) Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science, LNCS, vol. 903, pp. 38–50. Springer-Verlag, Herrsching, Germany (1994)
74. Ebert, J., Riediger, V., Winter, A.: Graph Technology in Reverse Engineering: The TGraph Approach. In: Proceedings of the 10th Workshop on Software Reengineering, pp. 67–81. Bad Honnef, Germany (2008)
75. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In: T. Pellegrini, S. Auer, K. Tochtermann, S. Schaffert (eds.) Networked Knowledge - Networked Media, pp. 7–24. Springer (2009)
76. Etcheverry, L., Vaisman, A.A.: QB4OLAP: A New Vocabulary for OLAP Cubes on the Semantic Web. In: Proceedings of the Third International Workshop on Consuming Linked Data (COLD). Boston, MA, USA (2012)

77. Filali, I., Bongiovanni, F., Huet, F., Baude, F.: A Survey of Structured P2P Systems for RDF Data Storage and Retrieval. In: Transactions on Large-Scale Data- and Knowledge-Centered Systems III, pp. 20–55. Springer-Verlag (2011)
78. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. Journal of Computer and System Sciences **31**(2), 182–209 (1985)
79. Flesca, S., Furfaro, F., Pugliese, A.: A Framework for the Partial Evaluation of SPARQL Queries. In: Proceedings of the 2nd International Conference on Scalable Uncertainty Management, pp. 201–214. Naples, Italy (2008)
80. Forgy, C.L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence **19**(1), 17–37 (1982)
81. Franz, T., Schultz, A., Sizov, S., Staab, S.: TripleRank: Ranking SemanticWeb Data By Tensor Decomposition. In: Proceedings of the 8th International Semantic Web Conference, pp. 213–228. Chantilly, VA, USA (2009)
82. Gallego, M.A., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An Empirical Study of Real-World SPARQL Queries. In: USEWOD (2011)
83. Gandon, F., Schreiber, G.: RDF 1.1 XML Syntax, W3C Recommendation 25 February 2014. http://www.w3.org/TR/rdf-syntax-grammar/
84. Golder, S., Huberman, B.A.: The structure of collaborative tagging systems. ArXiv Computer Science e-prints (2005)
85. Görlitz, O., Sizov, S., Staab, S.: PINTS: Peer-to-Peer Infrastructure for Tagging Systems. In: Proceedings of the 7th International Workshop on Peer-to-Peer Systems (IPTPS). Tampa Bay, Florida, USA (2008)
86. Görlitz, O., Sizov, S., Staab, S.: Tagster - Tagging-Based Distributed Content Sharing. In: S. Bechhofer, M. Hauswirth, J. Hoffmann, M. Koubarakis (eds.) Proceedings of the 5th European Semantic Web Conference, vol. 5021, pp. 807–811. Springer-Verlag, Tenerife, Canary Islands, Spain (2008)
87. Görlitz, O., Staab, S.: Federated Data Management and Query Optimization for Linked Open Data. In: A. Vakali, L.C. Jain (eds.) New Directions in Web Data Management, Studies in Computational Intelligence, vol. 331, pp. 109–137. Springer-Verlag (2011)
88. Görlitz, O., Staab, S.: SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In: Proceedings of the 2nd International Workshop on Consuming Linked Data. Bonn, Germany (2011)
89. Görlitz, O., Thimm, M., Staab, S.: SPLODGE: Systematic Generation of SPARQL Benchmark Queries for Linked Open Data. In: P. Cudré-Mauroux, J. Heflin, E. Sirin, et al. (eds.) Proceedings of the 11th International Semantic Web Conference (ISWC), LNCS, pp. 116–132. Springer-Verlag, Boston, MA, USA (2012)
90. Gray, J.: Database and Transaction Processing Performance Handbook. In: The Benchmark Handbook for Database and Transaction Systems, pp. 1–15. Morgan Kaufmann (1993)
91. Grimnes, G.A., Edwards, P., Preece, A.: Instance based Clustering of Semantic Web Resources. In: Proceedings of the 5th European Semantic Web Conference, pp. 303–317. Springer-Verlag, Tenerife, Canary Islands, Spain (2008)
92. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics: Science, Services and Agents on the World Wide Web **3**(2–3), 158–182 (2005)
93. Haas, L., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing Queries across Diverse Data Sources. In: Proceedings of the 23rd International Conference on Very Large Data Bases, pp. 276–285. Athens, Greece (1997)

94. Haas, L.M., Schwarz, P.M., Kodali, P., Kotlar, E., Rice, J.E., Swope, W.C.: DiscoveryLink: A system for integrated access to life sciences data sources. IBM Systems Journal **40**(2), 489–511 (2001)
95. Halpin, H., Hayes, P.: When owl:sameAs isn't the same: An analysis of identity links on the semantic web. In: Proceedings of the WWW2010 workshop on Linked Data on the Web, LDOW2010 (2010)
96. Halpin, H., Robu, V., Shepherd, H.: The Complex Dynamics of Collaborative Tagging. In: Proceedings of the 16th International World Wide Web Conference - WWW '07, pp. 211–220. ACM Press, New York, New York, USA (2007)
97. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009), pp. 94–109. Chantilly, VA, USA (2009)
98. Harris, S., Seaborne, A.: SPARQL Query Language 1.1, W3C Recommendation 21 March 2013. http://www.w3.org/TR/sparql11-query/
99. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: Third Latin American Web Congress (LA-WEB'2005), pp. 71–80. IEEE Computer Society, Buenos Aires, Argentina (2005)
100. Harth, A., Hogan, A., Delbru, R., Umbrich, J., O'Riain, S., Decker, S.: SWSE: Answers Before Links! Proceedings of Semantic Web Challenge (2007)
101. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.U., Umbrich, J.: Data Summaries for On-Demand Queries over Linked Data. In: Proceedings of the 19th International Conference on World Wide Web, pp. 411–420. ACM, Raleigh, NC, USA (2010)
102. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data From The Web. In: Proceedings of the 6th International Semantic Web Conference, pp. 211–224. Busan, Korea (2007)
103. Hartig, O.: Zero-Knowledge Query Planning for an Iterator Implementation of Link Traversal Based Query Execution. In: G. Antoniou, M. Grobelnik, E. Simperl, et al. (eds.) Proceedings of 8th Extended Semantic Web Conference - ESWC'11, LNCS, vol. 6643, pp. 154–169. Springer-Verlag, Heraklion, Crete, Greece (2011)
104. Hartig, O., Bizer, C., Freytag, J.C.: Executing SPARQL Queries over the Web of Linked Data. In: Proceedings of the 8th International Semantic Web Conference, pp. 293–309. Chantilly, VA, USA (2009)
105. Hartig, O., Freytag, J.C.: Foundations of Traversal Based Query Execution over Linked Data. In: Proceedings of the 23rd ACM Conference on Hypertext and Social Media - HT '12, pp. 43–52. ACM Press, Milwaukee, WI, USA (2012)
106. Hartig, O., Heese, R.: The SPARQL Query Graph Model for Query Optimization. In: E. Franconi, M. Kifer, W. May (eds.) Proceedings of the 4th European Semantic Web Conference - ESWC'07, LNCS, vol. 4519, pp. 564–578. Springer-Verlag, Innsbruck, Austria (2007)
107. Hartig, O., Langegger, A.: A Database Perspective on Consuming Linked Data on the Web. Datenbank-Spektrum **10**(2), 57–66 (2010)
108. Hausenblas, M., Halb, W., Raimond, Y., Feigenbaum, L., Ayers, D.: SCOVO: Using Statistics on the Web of Data. In: Proceedings of the 6th European Semantic Web Conference (ESWC), pp. 708–722. Springer-Verlag, Heraklion, Crete, Greece (2009)
109. Hausenblas, M., Karnstedt, M.: Understanding Linked Open Data as a Web-Scale Database. In: Proceedings of the Second International Conference on

Advances in Databases, Knowledge, and Data Applications, pp. 56–61. IEEE, Menuires, France (2010)

110. Hayes, J., Gutierrez, C.: Bipartite Graphs as Intermediate Model for RDF. In: Proceedings of the Third International Semantic Web Conference - ISWC 2004, pp. 47–61. Springer-Verlag, Hiroshima, Japan (2004)
111. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. Synthesis Lectures on the Semantic Web: Theory and Technology **1**(1), 1–136 (2011)
112. Herman, I., Adida, B., Sporny, M., Birbeck, M.: RDFa 1.1 Primer - Second Edition, Rich Structured Data Markup for Web Documents, W3C Working Group Note 22 August 2013. http://www.w3.org/TR/rdfa-primer/
113. Hogan, A., Harth, A., Decker, S.: ReConRank: A Scalable Ranking Method for Semantic Web Data with Context. In: Proceedings of the 2nd Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS). Athens, Georgia, USA (2006)
114. Hong, W., Stonebraker, M.: Optimization of parallel query execution plans in XPRS. Distributed and Parallel Databases **1**, 9–32 (1993)
115. Horowitz, K., Malkhi, D.: Estimating network size from local information. Inf. Process. Lett. **88**(5), 237–243 (2003)
116. Hose, K., Schenkel, R., Theobald, M., Weikum, G.: Database Foundations for Scalable RDF Processing. In: A. Polleres, C. d'Amato, M. Arenas, et al. (eds.) Proceedings of the 7th International Summer School on Reasoning Web: Semantic Technologies for the Web of Data, LNCS, vol. 6848, pp. 202–249. Springer-Verlag, Galway, Ireland (2011)
117. Hotho, A., Jäschke, R., Schmitz, C., Stumme, G.: Information retrieval in folksonomies: Search and ranking. In: Y. Sure, J. Domingue (eds.) The Semantic Web: Research and Applications, LNAI, vol. 4011, pp. 411–426. Springer, Heidelberg (2006)
118. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. In: Proceedings of the 37th International Conference on Very Large Data Bases - VLDB '11. Seattle, WA, USA (2011)
119. Husain, M.F., Doshi, P., Khan, L., Thuraisingham, B.: Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce. In: Proceedings of the 1st International Conference on Cloud Computing, pp. 680–686. Springer-Verlag, Beijing, China (2009)
120. Ioannidis, Y.: The History of Histograms (abridged). In: Proceedings of the 29th International Conference on Very Large Data Bases, vol. 29, pp. 19–30. VLDB Endowment, Berlin, Germany (2003)
121. Ioannidis, Y.E., Kang, Y.C.: Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. ACM SIGMOD Record **20**(2), 168–177 (1991)
122. Isaac, A., Summers, E.: SKOS Simple Knowledge Organization System Primer, W3C Working Group Note 18 August 2009. http://www.w3.org/TR/2009/NOTE-skos-primer-20090818/
123. Jelasity, M., Montresor, A.: Epidemic-style proactive aggregation in large overlay networks. In: Proceedings of The 24th International Conference on Distributed ComputingSystems (ICDCS), pp. 102–109. IEEE Computer Society, Tokyo, Japan (2004)

124. Kämpgen, B., Harth, A.: Transforming Statistical Linked Data for Use in OLAP Systems. In: Proceedings of the 7th International Conference on Semantic Systems - I-Semantics '11, pp. 33–40. ACM Press, Graz, Austria (2011)
125. Kaoudi, Z., Koubarakis, M., Kyzirakos, K., Miliaraki, I., Magiridou, M., Papadakis-Pesaresi, A.: Atlas: Storing, updating and querying RDF(S) data on top of DHTs. Web Semantics: Science, Services and Agents on the World Wide Web **8**(4), 271–277 (2010)
126. Kaoudi, Z., Kyzirakos, K., Koubarakis, M.: SPARQL Query Optimization on Top of DHTs. In: P.F. Patel-Schneider, Y. Pan, P. Hitzler, et al. (eds.) Proceedings of the 9th International Semantic Web Conference, pp. 418–435. Springer-Verlag, Shanghai, China (2010)
127. Karnstedt, M., Sattler, K.U., Hauswirth, M.: Scalable distributed indexing and query processing over Linked Data. Web Semantics: Science, Services and Agents on the World Wide Web **10**, 3–32 (2012)
128. Karnstedt, M., Sattler, K.U., Hauswirth, M., Schmidt, R.: A DHT-based Infrastructure for Ad-hoc Integration and Querying of Semantic Data. In: Proceedings of the 12th International Database Engineering & Applications Symposium - IDEAS '08, pp. 19–28. ACM Press, Coimbra, Portugal (2008)
129. Karnstedt, M., Sattler, K.U., Richtarsky, M., Müller, J., Hauswirth, M., Schmidt, R., John, R.: UniStore: Querying a DHT-based Universal Storage. In: Proceedings of the 23rd International Conference on Data Engineering - ICDE '07, pp. 1503–1504. IEEE, Istanbul, Turkey (2007)
130. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, p. 482. IEEE Computer Society, Washington, DC, USA (2003)
131. Konrath, M., Gottron, T., Staab, S., Scherp, A.: SchemEX – Efficient Construction of a Data Catalogue by Stream-based Indexing of Linked Data. Web Semantics: Science, Services and Agents on the World Wide Web **16**(5) (2012)
132. Kossmann, D.: The State of the Art in Distributed Query Processing. ACM Computing Surveys **32**(4), 422–469 (2000)
133. Kossmann, D., Stocker, K.: Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. ACM Transactions on Database Systems **25**(1), 43–82 (2000)
134. Kostoulas, D., Psaltoulis, D., Gupta, I., Birman, K., Demers, A.: Decentralized Schemes for Size Estimation in Large and Dynamic Groups. In: Proceedings of the 4th International Symposium on Network Computing and Applications, pp. 41–48. IEEE Computer Society, Washington, DC, USA (2005)
135. Ladwig, G., Tran, T.: Linked Data Query Processing Strategies. In: P.F. Patel-Schneider, Y. Pan, P. Hitzler, et al. (eds.) Proceedings of the 9th International Semantic Web Conference, LNCS, vol. 6496, pp. 453–469. Springer-Verlag, Shanghai, China (2010)
136. Ladwig, G., Tran, T.: SIHJoin: Querying Remote and Local Linked Data. In: G. Antoniou, M. Grobelnik, E. Simperl, et al. (eds.) Proceedings of 8th Extended Semantic Web Conference - ESWC'11, LNCS, vol. 6643, pp. 139–153. Springer-Verlag, Heraklion, Crete, Greece (2011)
137. Lakshman, A., Malik, P.: Cassandra - A Decentralized Structured Storage System. ACM SIGOPS Operating Systems Review **44**(2), 35 (2010)
138. Lambiotte, R., Ausloos, M.: Collaborative Tagging as a Tripartite Network. In: V.N. Alexandrov, G.D. van Albada, P.M. Sloot, J. Dongarra (eds.) Proceedings

of the 6th International Conference on Computational Science - ICCS ’06, vol. 3993, pp. 1114–1117. Springer-Verlag, Reading, UK (2006)
139. Langegger, A., Wöß, W.: RDFStats - An Extensible RDF Statistics Generator and Library. In: Proceedings of the 20th International Workshop on Database and Expert Systems Application, pp. 79–83. IEEE Computer Society, Linz, Austria (2009)
140. Langegger, A., Wöß, W., Blöchl, M.: A Semantic Web Middleware for Virtual Data Integration on the Web. In: Proceedings of the 5th European Semantic Web Conference, pp. 493–507. Tenerife, Canary Islands, Spain (2008)
141. Ley, M.: The DBLP computer science bibliography: Evolution, research issues, perspectives. In: Alberto H. F. Laender, A.L. Oliveira (eds.) Proceedings of the 9th International Symposium on String Processing and Information Retrieval, LNCS, vol. 2476, pp. 1–10. Springer-Verlag, Lisbon, Portugal (2002)
142. Li, Y., Heflin, J.: Using Reformulation Trees to Optimize Queries over Distributed Heterogeneous Sources. In: Proceedings of the 9th International Semantic Web Conference (ISWC), LNCS, vol. 6496, pp. 502–517. Springer-Verlag, Shanghai, China (2010)
143. Liarou, E., Idreos, S., Koubarakis, M.: Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In: Proceedings of the 5th International Semantic Web Conference, LNCS, vol. 4273, pp. 399–413. Springer-Verlag, Athens, GA, USA (2006)
144. Liarou, E., Idreos, S., Koubarakis, M.: Continuous RDF query processing over DHTs. In: Proceedings of the 6th International Semantic Web Conference, pp. 324–339. Springer-Verlag, Busan, Korea (2007)
145. Löser, A., Staab, S., Tempich, C.: Semantic Social Overlay Networks. IEEE Journal on Selected Areas in Communications **25**(1), 5–14 (2007)
146. Luo, Y., Lange, Y.D., Fletcher, G.H., De Bra, P., Hiders, J., Wu, Y.: Bisimulation Reduction of Big Graphs on MapReduce. In: Proceedings of the 29th British National Conference on Databases (BNCOD’13). Oxford, UK (2013)
147. Lynden, S., Kojima, I., Matono, A., Tanimura, Y.: ADERIS: An adaptive query processor for joining federated SPARQL endpoints. In: On the Move to Meaningful Internet Systems (OTM 2011), pp. 808–817. Springer-Verlag, Hersonissos, Crete, Greece (2011)
148. Lynden, S., Mukherjee, A., Hume, A.C., a.a. Fernandes, A., Paton, N.W., Sakellariou, R., Watson, P.: The design and implementation of OGSA-DQP: A service-based distributed query processor. Future Generation Computer Systems **25**(3), 224–236 (2009)
149. Mackert, L.F., Lohman, G.M.: R* Optimizer Validation and Performance Evaluation for Distributed Queries. In: Proceedings of the 12th International Conference on Very Large Data Bases - VLDB’86, pp. 149–159. ASM, Kyoto, Japan (1986)
150. Maduko, A., Anyanwu, K., Sheth, A., Schliekelman, P.: Graph Summaries for Subgraph Frequency Estimation. In: Proceedings of the 5th European Semantic Web Conference. Tenerife, Canary Islands, Spain (2008)
151. Magliacane, S., Bozzon, A., Valle, E.D.: Efficient Execution of Top-K SPARQL Queries. In: Proceedings of the 11th International Semantic Web Conference, pp. 344–360. Boston, MA, USA (2012)
152. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press (2008)

153. Mannino, M.V., Chu, P., Sager, T.: Statistical Profile Estimation in Database Systems. ACM Computing Surveys **20**(3), 191–221 (1988)
154. Manola, F., Miller, E.: RDF Primer, W3C Recommendation 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/
155. Massoulié, L., Merrer, E.L., Kermarrec, A., Ganesh, A.: Peer counting and sampling in overlay networks: random walk methods. In: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing. ACM, New York, NY, USA (2006)
156. McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language – Overview, W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-features/
157. Miranker, D.P., Depena, R.K., Jung, H., Sequeda, J.F., Reyna, C.: Diamond: A SPARQL Query Engine, for Linked Data Based on the Rete Match. In: Proceedings of the Artificial Intelligence meets the Web of Data workshop. Montpellier, France (2012)
158. Mirizzi, R., Ragone, A., Di Noia, T., Di Sciascio, E.: Ranking the Linked Data: The Case of DBpedia. In: B. Benatallah, F. Casati, G. Kappel, G. Rossi (eds.) Proceedings of the 10th International Conference on Web Engineering (ICWE), pp. 337–354. Springer-Verlag, Vienna, Austria (2010)
159. Mishra, P., Eich, M.H.: Join Processing in Relational Databases. ACM Computing Surveys **24**(1), 63–113 (1992)
160. Mitzenmacher, M.: Compressed bloom filters. IEEE/ACM Transactions on Networking **10**(5), 604–612 (2002)
161. Moerkotte, G., Neumann, T.: Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 930–941. Seoul, Korea (2006)
162. Möller, K., Hausenblas, M., Cyganiak, R., Grimnes, G.A., Handschuh, S.: Learning from Linked Open Data Usage: Patterns & Metrics. In: Proceedings of the Web Science Conference, pp. 1–8 (2010)
163. Montoya, G., Vidal, M.E., Acosta, M.: A Heuristic-Based Approach for Planning Federated SPARQL Queries. In: Proceedings of the Third International Workshop on Consuming Linked Data - COLD'12. Boston, MA, USA (2012)
164. Montoya, G., Vidal, M.E., Corcho, O., Ruckhaus, E., Buil-Aranda, C.: Benchmarking Federated SPARQL Query Engines: Are Existing Testbeds Enough? In: P. Cudré-Mauroux, J. Heflin, E. Sirin, et al. (eds.) Proceedings of the 11th International Semantic Web Conference (ISWC), LNCS, pp. 313–324. Springer-Verlag, Boston, MA, USA (2012)
165. Montresor, A., Jelasity, M.: PeerSim: A scalable P2P simulator. In: Proceedings of the 9th International Conference on Peer-to-Peer Computing, pp. 99–100. IEEE, Seattle, WA, USA (2009)
166. Mora, O., Engelbrecht, G., Bisbal, J.: A service-oriented distributed semantic mediator: integrating multiscale biomedical information. IEEE Transactions on Information Technology in Biomedicine **16**(6), 1296–1303 (2012)
167. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.: Usage-Centric Benchmarking of RDF Triple Stores. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, vol. 2. Toronto, Ontario, Canada (2012)
168. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In:

Proceedings of the 10th International Semantic Web Conference - ISWC ’11, LNCS, vol. 7031, pp. 454–469. Springer (2011)
169. Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmér, M., Risch, T.: EDUTELLA: P2P Networking for the Semantic Web. In: Proceedings of the 11th International World Wide Web Conference - WWW ’02, pp. 604–615. ACM Press, Hawaii, USA (2002)
170. Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M., Brunkhorst, I., Löser, A.: Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In: Proceedings of the 12th International World Wide Web Conference - WWW ’03, pp. 536–543. ACM Press, Budapest, Hungary (2003)
171. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In: 27th International Conference on Data Engineering (ICDE), pp. 984–994. IEEE, Hannover, Germany (2011)
172. Neumann, T., Weikum, G.: RDF-3X: a RISC-style Engine for RDF. In: Proceedings of the 34th International Conference on Very Large Data Bases, pp. 647–659. Auckland, New Zealand (2008)
173. Neumann, T., Weikum, G.: x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. Proceedings of the VLDB Endowment **3**(1-2), 256–263 (2010)
174. Newman, R.: Tag ontology design. http://holygoat.co.uk/projects/tags/ (2005). Last updated 2005-03-29
175. Nixon, L.J.B., Simperl, E., Krummenacher, R., Martin-Recuerda, F.: Tuplespace-based computing for the Semantic Web: a survey of the state-of-the-art. The Knowledge Engineering Review **23**(2), 181–212 (2008)
176. Ntarmos, N., Triantafillou, P., Weikum, G.: Counting at large: Efficient cardinality estimation in internet-scale data networks. In: Proceedings of the 22nd International Conference on Data Engineering, p. 40. IEEE Computer Society, Washington, DC, USA (2006)
177. Obermeier, P., Nixon, L.: A Cost Model for Querying Distributed RDF-Repositories with SPARQL. In: F. van Harmelen, A. Herzig, P. Hitzler, et al. (eds.) Proceedings of the Workshop on Advancing Reasoning on the Web: Scalability and Commonsense. Tenerife, Canary Islands, Spain (2008)
178. Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice.com: A Document-oriented Lookup Index for Open Linked Data. International Journal of Metadata, Semantics and Ontologies **3**(1), 37–52 (2008)
179. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 3rd Ed. Springer (2011)
180. Page, K., De Roure, D., Martinez, K., Sadler, J., Kit, O.: Linked Sensor Data: RESTfully serving RDF and GML. In: Proceedings of the 2nd International Workshop on Semantic Sensor Networks (SSN09), pp. 49–63. CEUR Workshop Proceedings, Washington DC, USA (2009)
181. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Tech. rep. (1999)
182. Patni, H., Henson, C., Sheth, A.: Linked Sensor Data. In: Proceedings of the International Symposium on Collaborative Technologies and Systems, pp. 362–370. IEEE, Chicago, IL, USA (2010)
183. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems **34**(3), 1–45 (2009)

184. Picalausa, F., Vansummeren, S.: What are real SPARQL queries like? In: Proc. of the International Workshop on Semantic Web Information Management, pp. 1–6 (2011)
185. Prasser, F., Kemper, A., Kuhn, K.A.: Efficient Distributed Query Processing for Autonomous RDF Databases. In: Proceedings of the 15th International Conference on Extending Database Technology - EDBT '12, pp. 372–383. ACM Press, Berlin, Germany (2012)
186. Prud'hommeaux, E., Buil-Aranda, C.: SPARQL 1.1 Federated Query, W3C Recommendation 21 March 2013. http://www.w3.org/TR/sparql11-federated-query/
187. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF, W3C Recommendation 15 January 2008. http://www.w3.org/TR/rdf-sparql-query/
188. Quilitz, B., Leser, U.: Querying Distributed RDF Data Sources with SPARQL. In: Proceedings of the 5th European Semantic Web Conference, pp. 524–538. Tenerife, Canary Islands, Spain (2008)
189. Rakhmawati, N.A., Hausenblas, M.: On the Impact of Data Distribution in Federated SPARQL Queries. In: Sixth International Conference on Semantic Computing - ICSC'12, pp. 255–260. IEEE, Palermo, Italy (2012)
190. Rakhmawati, N.A., Umbrich, J., Karnstedt, M., Hasnain, A., Hausenblas, M.: Querying over Federated SPARQL Endpoints – A State of the Art Survey. Tech. rep. (2013)
191. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A Scalable Content Addressable Network. In: Proceedings of the ACM SIGCOMM, pp. 161–172. San Diego, CA, USA (2001)
192. Rodriguez, M.A.: A Graph Analysis of the Linked Data Cloud. Arxiv preprint arXiv:0903.0194 pp. 1–7 (2009)
193. Rohloff, K., Schantz, R.E.: High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: The SHARD Triple-Store. In: Programming Support Innovations for Emerging Distributed Applications on - PSI EtA '10, pp. 1–5. ACM Press, Reno, NV, USA (2010)
194. Roth, M.T., Özcan, F., Haas, L.M.: Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In: Proceedings of the 25th International Conference on Very Large Data Bases - VLDB'99, pp. 599–610. Morgan Kaufmann, Edinburgh, Scotland, UK (1999)
195. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware, pp. 329–350 (2001)
196. Schenk, S., Staab, S.: Networked Graphs: A Declarative Mechanism for SPARQL Rules, SPARQL Views and RDF Data Integration on the Web. In: Proceeding of the 17th International World Wide Web Conference, pp. 585–594. Beijing, China (2008)
197. Schlosser, M., Sintek, M., Decker, S., Nejdl, W.: HyperCuP – Hypercubes, Ontologies, and Efficient Search on Peer-to-Peer Networks. In: First International Workshop Agents and Peer-to-Peer, pp. 112–124. Bologna, Italy (2003)
198. Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In: Proceedings of the 10th International Semantic Web Conference (ISWC), pp. 585–600. Springer-Verlag (2011)

199. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^{2}Bench: A SPARQL Performance Benchmark. In: Proceedings of the 25th International Conference on Data Engineering (ICDE), pp. 222–233. Shanghai (2009)
200. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Proceedings of the 13th International Conference on Database Theory - ICDT '10, pp. 4–33. ACM Press, Lausanne, Switzerland (2010)
201. Schmitz, C., Hotho, A., Jäschke, R., Stumme, G.: Mining association rules in folksonomies. In: Proceedings of the 10th Conference on Data Science and Classification IFCS, pp. 261–270. Springer, Ljubljana (2006)
202. Schreiber, G., Raimond, Y.: RDF 1.1 Primer, W3C Working Group Note 24 June 2014. http://www.w3.org/TR/rdf11-primer/
203. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization Techniques for Federated Query Processing on Linked Data. In: L. Aroyo, C. Welty, H. Alani, et al. (eds.) Proceedings of the 10th International Semantic Web Conference (ISWC), LNCS, vol. 7031, pp. 601–616. Springer-Verlag, Bonn, Germany (2011)
204. Schwarz, H., Ebert, J.: Bridging Query Languages in Semantic and Graph Technologies. In: Proceedings of the 6th International Reasoning Web Summer School on Semantic Technologies for Software Engineering, pp. 119–160. Dresden, Germany (2010)
205. Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T.: Access Path Selection in a Relational Database Management System. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 23–34. Boston, MA, USA (1979)
206. Sheth, A., Larson, J.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Surveys **22**(3), 183–236 (1990)
207. Staab, S., Stuckenschmidt, H. (eds.): Semantic Web and Peer-to-Peer: Decentralized Management and Exchange of Knowledge and Information. Springer (2006)
208. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In: Proceeding of the 17th International Conference on World Wide Web, pp. 595–604. ACM Press, Beijing, China (2008)
209. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup protocol for internet applications. In: Proceedings of the ACM SIGCOMM, pp. 149–160. San Diego, CA, USA (2001)
210. Stuckenschmidt, H., Vdovjak, R., Broekstra, J., Houben, G.J.: Towards Distributed Processing of RDF Path Queries. International Journal of Web Engineering and Technology **2**(2/3), 207–230 (2005)
211. Stuckenschmidt, H., Vdovjak, R., Houben, G.J., Broekstra, J.: Index Structures and Algorithms for Querying Distributed RDF Repositories. In: Proceedings of the 13th International World Wide Web Conference, pp. 631–639. New York, NY, USA (2004)
212. Stutz, P., Bernstein, A., Cohen, W.: Signal/Collect: Graph Algorithms for the (Semantic) Web. In: P.F. Patel-Schneider, Y. Pan, P. Hitzler, et al. (eds.) Proceedings of the 9th International Semantic Web Conference, ISWC 2010, pp. 764–780. Springer-Verlag, Shanghai, China (2010)

213. Svoboda, M., Mlýnková, I.: Linked Data Indexing Methods: A Survey. In: Proceedings of the 7th International IFIP Workshop on Semantic Web & Web Semantics, pp. 474–483. Springer-Verlag, Crete, Greece (2011)
214. Tempich, C., Staab, S., Wranik, A.: REMINDIN': Semantic Query Routing in Peer-to-Peer Networks Based on Social Metaphors. In: Proceedings of the 13th World Wide Web conference - WWW '04, pp. 640–649. ACM Press, New York, NY, USA (2004)
215. Tran, T., Ladwig, G.: Structure Index for RDF Data. In: Workshop on Semantic Data Management (2010)
216. Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V., Boncz, P.: Heuristics-based Query Optimisation for SPARQL. In: Proceedings of the 15th International Conference on Extending Database Technology - EDBT '12, pp. 324–335. ACM Press, Berlin, Germany (2012)
217. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: A Graph Based RDF Index. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence, pp. 1465–1470. AAAI Press, Vancouver, Canada (2007)
218. Umbrich, J., Hogan, A., Polleres, A., Decker, S.: Improving the Recall of Live Linked Data Querying through Reasoning. In: M. Krötzsch, U. Straccia (eds.) Proceedings of the 6th International Conference on Web Reasoning and Rule Systems, LNCS, vol. 7497, pp. 188–204. Vienna, Austria (2012)
219. Umbrich, J., Hose, K., Karnstedt, M., Harth, A., Polleres, A.: Comparing data summaries for processing live queries over Linked Data. World Wide Web Journal **14**(5–6), 495–544 (2011)
220. Urhan, T., Franklin, M.J.: XJoin: A Reactively-Scheduled Pipelined Join Operator. The Bulletin of the Technical Committee on Data Engineering **23**(2), 1–7 (2000)
221. Vandervalk, B.P., McCarthy, E.L., Wilkinson, M.D.: Optimization of Distributed SPARQL Queries Using Edmonds' Algorithm and Prim's Algorithm. In: Proceedings of the 12th International Conference on Computational Science and Engineering, pp. 330–337. IEEE Computer Society, Vancouver, Canada (2009)
222. Vandervalk, B.P., McCarthy, E.L., Wilkinson, M.D.: SHARE: A Semantic Web Query Engine for Bioinformatics. In: A. Gómez-Pérez, Y. Yu, Y. Ding (eds.) Proceedings of the Fourth Asian Semantic Web Conference, ASWC 2009, LNCS, vol. 5926, pp. 367–369. Springer-Verlag, Shanghai, China (2009)
223. W3C OWL Working Group: OWL 2 Web Ontology Language, Document Overview (Second Edition), W3C Recommendation 11 December 2012. http://www.w3.org/TR/owl2-overview/
224. Walavalkar, O., Joshi, A., Finin, T., Yesha, Y.: Streaming Knowledge Bases. In: Proceedings of the 4th International Workshop on Scalable Semantic Web knowledge Base Systems - SSWS'08. Karlsruhe, Germany (2008)
225. Wang, X., Tiropanis, T., Davis, H.C.: Evaluating Graph Traversal Algorithms for Distributed SPARQL Query Optimization. In: J.Z. Pan, H. Chen, H.G. Kim, et al. (eds.) Proceedings of the Joint International Semantic Technology Conference - JIST '11, LNCS, vol. 7185, pp. 210–225. Springer-Verlag, Hangzhou, China (2011)
226. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. In: Proceedings of the 34th International Conference on Very Large Data Bases, pp. 1008–1019. Auckland, New Zealand (2008)

227. White, T.: Hadoop: The definitive guide. O'Reilly Media (2012)
228. Wiederhold, G.: Mediators in the Architecture of Future Information Systems. Computer **25**(3), 38–49 (1992)
229. Wilkinson, K.: Jena Property Table Implementation. In: Proceedings of the Second International Workshop on Scalable Semantic Web Knowledge Base Systems - SSWS'06 (2006)
230. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In: I.F. Cruz, V. Kashyap, S. Decker, R. Eckstein (eds.) Proceedings of the First International Workshop on Semantic Web and Databases - SWDB'03, pp. 131–150. Berlin, Germany (2003)
231. Wilschut, A.N., Apers, P.M.: Dataflow query execution in a parallel main-memory environment. Distributed and Parallel Databases **1**(1), 103–128 (1993)
232. Yang, S., Yan, X., Zong, B., Khan, A.: Towards Effective Partition Management for Large Graphs. In: Proceedings of the 2012 International Conference on Management of Data - SIGMOD '12, pp. 517–528. ACM Press, Scottsdale, AZ, USA (2012)
233. Zemánek, J., Schenk, S., Svátek, V.: Optimizing SPARQL Queries over Disparate RDF Data Sources through Distributed Semi-joins. In: Poster & Demos of the International Semantic Web Conference - ISWC'08 (2008)
234. Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: gStore: Answering SPARQL Queries Via Subgraph Matching. In: Proceedings of the 37th International Conference on Very Large Data Bases - VLDB '11, vol. 4, pp. 482–493. Seattle, WA, USA (2011)

Index

Adaptive Query Processing, 77
Aggregates, 20
AllegroGraph, 28

Backlinks, 16, 17
Billion Triple Challenge, 145
Bind Join, 76
Bisimulation, 24
Blank Node, 10, 14
Bloom Filter, 60, 76
Bloom Join, 76
BSBM, 131
Bushy Trees, 74

Characteristic Sets, 60, 142
ChEBI, 132
CKAN, 4, 39, 54, 58
Collaborative Tagging, 108
CORDIS, 3
Cosine Similarity, 111
CPU Cost, 92

D2RServer, 25
DBLP, 3, 131
DBpedia, 3, 16, 132
DBPSB, 132
DHT, 116
Distributed HashTable, 48
Document-centric Search, 37
Drugbank, 90, 132
Dublin Core, 12
Dynamic Programming, 77
Dynamic Source Discovery, 54

Edutella, 47
Entity-centric Search, 37
Equi-Join, 97
Event notifications, 51
Event subscriptions, 51
Extract-Transform-Load, 35

Falcons, 36
FedBench, 66, 132
Fetch Matches, 75
Flooding, 47
FOAF, 3, 12
Folksonomy, 109
Functional Requirements, 30

Graph Database, 27
GridVine, 49

Hash Join, 75
Hash Sketch, 60
Histogram, 59
HyperCuP, 47
Hypergraph, 109

Independent Attributes, 94, 95, 97, 98
Instance Index, 41
Inverse Document Frequency, 112
ISWC, 145
Iterative Dynamic Programming, 77
Iterator-based Query Evaluation, 72

Join order optimization, 81

Language tag, 10
Left-deep Tree, 73
LIDAQ, 132

Link Traversal Query Processing, 44
Linked Data Cloud, 16
Linked Data Patterns, 2
Linked Open Data, 1, 14
Literal, 10, 59
LUBM, 131

MapReduce, 36
MicroData, 1
Microformats, 1, 13
MusicBrainz, 132

N3, 13
Named Graph, 13
Neo4J, 28
Nested-Loop Join, 74
Non-blocking Iterator, 44
Non-functional Requirements, 31
NQuads, 147

OLAP, 35
Ontology, 58
Open Knowledge Foundation, 145
Overlay Network, 47
OWL, 11

Peer-to-peer, 47
PeerSim, 124
Property Table, 22
Publish/Subscribe, 51

Q-Tree, 60
Quads, 14
Quadstore, 23
Query Optimization Heuristics, 81

RDF, 2, 9
RDF Data Cube, 85
RDF Schema, 11
RDF/XML, 12, 14
RDFa, 13
RDFPeers, 49
RDMBS, 21
Rete, 45, 51

SameAs Links, 15
Schema Index, 40
Schema.org, 1
SCOVO, 85
Self-Join, 98
Semantic Overlay Networks, 47
Semi-Join, 75
Sesame, 66
Ship Whole, 75
Sindice, 36, 54
SKOS, 12
Sort-Merge Join, 75
SP^{2}Bench, 131
SPARQL, 17
  Ask Query, 18, 136
  Construct Query, 18, 136
  Describe Query, 18, 136
  Property Path, 20
  Select Query, 18, 136
  Service Keyword, 20
  Values Keyword, 20
SPARQL 1.1, 20
SPARQL endpoint, 2
SPARQL Federation, 20
SQL, 20, 26
Static Source Definitions, 54
Structure Index, 40
Sub Queries, 20
Super-Peers, 47
Swoogle, 36
SWSE, 36
Symmetric Hash Join, 76

Term Frequency, 112
TF-IDF, 108, 112
TriG, 14
Tripartite Network, 109
Triple Pattern, 17
Triplestore, 23
TriX, 14
Turtle, 12, 14

Uniform Distribution, 94, 96
URIrefs, 10
URL, 10

Vector Space Model, 111
VoiD, 4, 55, 85
  Class Partition, 55, 90
  Linkset, 91
  Property Partition, 55, 90

W3C, 9, 17, 20
Watson, 36
Web of Data, 14
Wikipedia, 3, 16
  Infoboxes, 16
World Wide Web, 14

Lebenslauf

Olaf Görlitz
Diplom-Informatiker

08/2005 - 12/2012	Wissenschaftlicher Mitarbeiter am Institute for Web Science and Technologies, Universität Koblenz-Landau
10/1999 - 06/2005	Studium der Informatik, Technische Universität Berlin Titel der Diplomarbeit: „Multipath Routing in Publish/Subscribe Systems“
08/2002 - 07/2003	Visiting Researcher, Internet Systems and Storage Group Duke University, Durham, NC, USA